October 2007

An SOA Approach To Sensor Services

Brett M. Levasseur
Worcester Polytechnic Institute

Keith Phillip Craig
Worcester Polytechnic Institute

MXC-L071

An SOA Approach

to

Sensor Services

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fufillment of the requirements for the

Degree of Bachelor of Science

by

Keith Craig and Brett Levasseur

Professor Michael J. Ciaraldi, Project Advisor

Abstract

The purpose of this project is to demonstrate the use of Semantic Web design and Service Oriented Architecture principles to make sensor information and data readily accessible to clients across the Internet. This project produced a set of XML files that described sensors, defined an ontology that described the vocabulary used in the XML files, and configured an existing registry technology for use within our sensor domain.

Acknowledgements

We would like to thank our supervisors Bill Moser and Oliver Newell at Lincoln Laboratory for their invaluable assistance in all facets of this project. We would additionally like to thank Farruk Najmi for his advice regarding ebXML Registries. We would also like to thank Professor Ciaraldi for his help on the WPI side of things.

The OWL files were created using the Protégé resource, which is supported by grant LM007885 from the United States National Library of Medicine.

Contents

List of Figures

List of Tables

1 Introduction

When we want the weather forecast, we can just turn on the television, listen to the radio, or visit our favorite weather website. In these places, the data appears, nicely processed for our consumption. The data, however, clearly doesn't start out this way. Meteorologists must get this data from numerous weather sensors around the country and analyze it to get the forecast that we all see. Meteorologists typically get this data along pre-determined channels.

This is a generally effective set-up, but how can it be improved? What happens if the meteorologists want information from a sensor that is not a part of their standard network? What if they need to change the setup of sensors that they pull information from? Do they spend the time and the money to write an entirely new computer program to contact the new sensors? Unless new sensors are needed for new coverage, this solution is rarely cost- or time-effective. Rather, it is likely sensors already exist in locations where they are needed. How can one make the most out of our pre-existing sensors? Is there a system that will allow for easy discovery of and connection to these sensors? It turns out that the idea of Service Oriented Architecture provides the approach needed.

Service Oriented Architecture allows an organization to break down its resources into a set of services. It can then publish these services to a registry that is publicly available. This registry allows for the search and discovery of the services by clients who need them. The registry also includes all of the information clients will need to connect (or *bind*) to the services so that they can be used. Now clients can find the weather sensors services they need and bind to them. If clients need to change the services they use or what sensors they get the services from, they can do so with minimal change to their existing programs.

2 Problem Statement

The members of MIT Lincoln Laboratory Group 43 - Weather Sensing - wanted to create a Service Oriented Architecture for weather sensors. This systems registry would provide information on weather sensors so that clients can discover the sensors and sensor services they need. The registry would also contain information to aid in the search and discovery of services by machines. To accomplish this last goal, Group 43 wanted to develop an ontology that would provide computers with information on the services and the logical relationships of the data.

One of the customers that can benefit from a system like this would be the Federal Aviation Administration (FAA). A weather sensor Service Oriented Architecture would allow the FAA to find the weather sensors and services they needed for a task quickly and efficiently. It would also give them the ability to use programs that take advantage of the modular nature of Service Oriented Architecture. Instead of forcing the FAA to develop new programs to connect to different sensors, they could just change the bindings within the programs they already have. The FAA has already demonstrated an interest in Service Oriented Architecture by investing in projects like the System Wide Information Management (SWIM) system to integrate the data they collect and make it available to other organizations.

Our project has four primary deliverables. The first is a set of files that describe weather sensors. These files should describe the sensors to a sufficient extent that a client receiving them would be able to determine all necessary features of the sensor from them. For our project, this means that each file should uniquely identify a sensor installation and its available services. Our advisors have provided us with information on a variety of sensors. It is from these that we worked.

The second deliverable for this project is a sensor ontology. When client machines run searches against our registry we cannot be sure how they will format their queries or what terminology they use. By creating an ontology we will be providing resources needed to have computer agents to find the logical correlations between the clients' queries and our sensor information. This idea of providing both human and computer interpretable data has roots in a concept called the Semantic Web.

The third deliverable is a registry and repository. This will hold our sensor and ontology files. This registry is the database against which clients will run queries. The registry will hold data relevant to potential client queries, while the associated repository will hold information that, while not necessarily directly related to searching, will be useful for connecting and identifying specific characteristics about the sensors.

Our fourth deliverable is a test client. The goal of this project is to create a working example of a registry for weather sensors. However, without a client to run test queries we will not know if our registry is designed properly. We can identify different types of questions that our registry should be able to answer. We can also tell what sensors apply to our different queries. Time permitting, this client might also serve as a demonstration of potential things that can be done given the results that the registry can return. For example, we may translate the search results into a Google Earth KML file that can be used to graphically display the locations of discovered sensors.

The members of Group 43 at Lincoln Laboratory have given us this project so that they may see an example of Service Oriented Architecture and how to develop it. Other businesses and educational facilities have performed their own research into these areas and produced their own examples. The members of Group 43, however, would like to see an example based

on their own sensors and using the technologies and standards that they want to use.

3 Background

Our project was not conceived in a vacuum, but rather it leveraged several pre-existing technologies and ideas relating to web-based data storage and representation.

3.1 Service Oriented Architecture

The idea of Service Oriented Architecture formed out of a need for more flexible, business-oriented systems. Usually a business will have many different systems designed to handle different tasks and provide services. Many of these systems, however, are not networked together to allow their usage by many groups and users. Additionally, all too often these systems are also incompatible with one another since they were made by different people, at different times, and for different purposes. Instead of this, businesses wanted services that were oriented towards departments and business units. In this design we have many services that represent the different aspects of the business. The systems that we then make to support the business needs make use of these individual components that are modifiable and reusable when needed[4].

Service Oriented Architecture is a form of distributed computing where service providers publish their services to a registry where they may be easily discovered by clients who need them. This registry contains information on how to connect to the service and what data the service needs to operate. Additionally, a Service Oriented Architecture aims to be implementation-, architecture-, and language-agnostic. A company can provide (for example) a currency conversion service and other companies can use it without worrying about whether it was implemented in Java, C++, or .NET on a SPARC, x86, or PowerPC archi-

tecture running BSD, Unix, or Windows.

Many business have already begun to take advantage of Service Oriented Architecture. Retailers are sharing data between multiple stores faster and more efficiently than before. Retail businesses will often have services such as payment processing, inventory, ordering, return policies, create-invoice, and others. All of these can be made into services rather than different non-interconnected systems[10]. One company called Accenture has developed systems to allow government revenue agencies to make services available to people and businesses to facilitate quick access to documents, information, and services. In Australia, Accenture created a system for the Australian Tax Office that allowed businesses to request their own unique identifiers needed for tax invoices. The Service Oriented Architecture allowed this service to be completed in minutes instead of days[1].

Service Oriented Architecture allows many clients to search and find the services they need from multiple providers. An inherent problem with this is how all of these businesses represent their services. To avoid this problem, companies can use developed standards that everyone can follow. To this end, the Open Geospatial Consortium has developed Sensor Web Enablement (SWE), which is a standard for developing sensor webs. The OGC is an organization made up of companies, government agencies, and universities that produce standards for technologies that add spatial data to computing services such as applications or the Web. The SWE standard provides the resources and the guidance needed when developing a set of sensors interconnected using the Web. The current listing of specifications in the SWE include:

1. Observations & Measurements (O&M) - Standard models and XML schema for encoding observations and measurements from a sensor, both archived and real-time.

2. Sensor Model Language (SensorML) - Standard models and XML schema for describing sensor systems and processes; provides information needed for discovery of sensors, location of sensor observations, processing of low-level sensor observations, and listing of taskable properties.
3. Transducer Model Language (TML) - The conceptual model and XML schema for describing transducers and supporting real-time streaming of data to and from sensor systems.
4. Sensor Observations Service (SOS) - Standard web service interface for requesting, filtering, and retrieving observations and sensor system information. This is the intermediary between a client and an observation repository or near real-time sensor channel.
5. Sensor Planning Service (SPS) - Standard web service interface for requesting user-driven acquisitions and observations. This is the intermediary between a client and a sensor collection management environment.
6. Sensor Alert Service (SAS) - Standard web service interface for publishing and subscribing to alerts from sensors.
7. Web Notification Services (WNS) - Standard web service interface for asynchronous delivery of messages or alerts from SAS and SPS web services and other elements of service workflows.

For the purposes of this project we will only be using the SensorML standard since we are only trying to develop a registry of sensors and not an entire system.

Our work for Lincoln Laboratory in the field of Service Oriented Architecture for sensors is not new. Many other companies and educational facilities have done their own research into this field. For example, Xingchen Chu and Rajkumar Buyya[6] from the University of Melbourne, Australia, developed a sensor web that made a set of sensors and their services discoverable and controllable over the Internet. While this area has been researched, we will also be adding our own input to the current collection of information.

In their paper, Xingchen and Rajkumar found that Sensor Registry via SensorML needs to be developed in order to support the worldwide sensor registration and discovery[6]. In fact, the OGC is working on a Sensor Registry Information Model to develop sensor descriptions

and sensor types and instances and publish sensor types and instances to an online instance of Sensor Registry Service[14]. Phase 3 of the OGC Web Services or OWS-3 started in 2005; since then the released OWS-4 does not include any information relating to the problem. OWS-4 does define some registry abilities such as the Sensor Alert Service described above, but this does not provide information on how to publish the SensorML files to a registry. Currently the OGC is developing the OWS-5 though it is not yet completed.

There are many benefits to using Service Oriented Architecture with sensors. When you make your sensors their own service providers you are promoting modularity in programming. Client programs can just connect to the service on the particular sensor or sensors they need. If a sensor or sensor service goes down, then it is relatively easy for a client to change their binding to use a different sensor that fits their requirements. This modularity also means that programs can remain useful longer since they can adapt to new sensors.

3.2 Semantic Web

Currently, the Internet is very good at providing information that humans can understand. Computers, however, do not understand the content they carry. This means that the human user has to put in the work to find and collect the information that is relevant to them. Search engines like Google or Yahoo allow users to type in what they want and return possible matches. These services can do a good job at this, however they are just using searching algorithms based on the text entered. Things would be much easier if these search engines and other computer agents could understand the concept of what the user wants rather than just performing functions on a string literal. This is where the Semantic Web comes in.

The World Wide Web Consortium and its founder Tim Berners-Lee developed the idea of the Semantic Web and its supporting standards and technologies. The Semantic Web seeks to provide computers with the ability to understand the data they carry. In other words, "[t]he Semantic Web will be an abstract representation of data on the Web"[2]. In order to develop a Semantic Web, the people who publish their information and content to the Web would also publish a vocabulary to define their words and concepts. These vocabularies can be hyperlinked together so that developers can use pre-existing vocabularies[2].

Most of the information on the Internet is defined using HTML that describes how the content should appear on a Web page. This allows a person to create content that is easy for a human user to read and understand. XML allows for better exchange of data across the Internet by giving the data better structure for *c*omputers to understand.

To produce a Semantic Web, we need to define the logical relationships among the concepts and information that we provide. This can be done using languages like RDF and OWL. RDF allows the user to describe their data in terms of subject-predicate-object expressions. OWL allows the user to define the logical relationships amongst the terms in a vocabulary to create an ontology[2].

So why is a Semantic Web so important? Why do we need to invest in this research? For the regular user, having computers that can understand information can make many tasks they perform simpler and more meaningful. The real importance, however, comes from the effect it could have on business. Businesses need information on the market, their suppliers, and competitors to help them decide what actions to take. The tools of the Semantic Web will give businesses the ability to better share and relate their information. Companies and organizations will be able to build better systems to manage and use their information[15].

Companies with physical stores can use Semantic Web systems to help provide them with the information they need to get their customers the products and services that they want. The Semantic Web will also have a major impact on Internet shopping. The website CIO.com, which is a group that provides analysis information on technology and technology trends for businesses, found that in 2005, consumer spending was $172 billion in North America alone[15]. With this number only expected to increase, businesses have an opportunity to become leaders in the online market if they can develop Semantic Web systems to gain an advantage over their competitors[15].

There are already companies who have developed Semantic systems to help themselves in business. The scientific publisher Elsevier found that they were getting multiple requests for information that went across many different publications. It became increasingly harder for the company to find all the relevant data especially since searching their computer data for particular words often returned less than meaningful results. To solve this Elsevier is attempting to make an ontology that indexes all of their publications to allow for better searching. Another example comes from the car manufacturer Audi. Currently Audi operates so many different databases that keeping track of them is nearly impossible. An inevitable consequence of this is that some information gets needlessly duplicated while other information is left out entirely. Audi is now using ontologies to combine these multiple sources of data into one. By the nature of mapping to the data, the old databases can continue to be used while the new system is being developed[3].

For our project we want to make sure that the most number of clients possible can have access to and understand the data and services in our registry. To achieve this we are adding a Semantic Web aspect to our project through use of our sensor ontology. This ontology

will allow clients to be able to find the relationships between its queries and the information available in the registry. This way our registry will be able to produce more meaningful results to the client and make our registry easier to use.

3.3 Ontology

In philosophy, ontology refers to the study of existence and seeks to understand the nature of things that exist. In computer science, ontology refers to the use of a data schema (called an ontology) to define and link concepts within a certain scope. Ontology generally concerns itself with the following aspects of its scope: individuals, classes, attributes, relations, and events.

Individuals represent the lowest level components of the ontology. These are the specific object instances that the ontology is supposed to classify. While Ontology may not specifically delineate individual members, they are nevertheless the underlying objects.

Classes represent groups of individuals or other classes linked through some common aspect. For example, an ontology focused on vehicles might have the classes *Car* and *Truck* representing all the individual vehicles that would fall into each of those categories.

Attributes describe some feature of an individual or a class. For instance, in the example of a vehicle ontology, there might be an attribute called "Wheels", the parameter of which would simply be how many wheels a vehicle had.

Relations describe the connections between classes, individuals, or attributes. They are typically represented as an attribute with another object in the ontology as a parameter. Common relations include "subclass-of" and "part-of".

Related to the subclass relation is the concept of a partition. A partition is simply a set

of classes that are all subclasses of the same superclass, in addition to the rules that define the separation of the subclasses. If all the subclasses in a partition are mutually exclusive, the partition is *disjoint*. If the partition is such that all individuals in the superclass are in a subclass in the partition, the partition is *exhaustive*.

Ontologies are not a new concept to computer science. In the field of artificial intelligence, ontologies are seen as a way to provide systems with the knowledge they need to be intelligent. With an ontology an intelligent system has defined for them "what there is in the world, which are the properties and relations of things, processes and events, what are the causal connections, what are the laws, etc"[16].

The development of ontology is a very important step. It would be far too easy to come up with a simple document defining the hierarchy of a system and some relations. An ontology has the responsibility to be more than that. It has to accurately define concepts and objects in a way to best fit how they will be used. Take the example of ontology for buying a computer compared to one that describes the components of a computer. In the ontology for a computer buying system you may have your classes as being different types of computers with their components being attributes. On the other hand if your ontology is meant to describe a computer in general then it may be more beneficial to make all of the components of the computer classes with more specific attributes. Research done into ontology development has led to some guidelines that we will follow.

Our project will be adding to the current body of ontologies by making one for the sensors that we will have available in our registry. While some ontologies do already exist, they do not all seem to take the full advantage that the technology has to offer. The Semantic Web for Earth and Environmental Terminology (SWEET) set that was developed by NASA are

actually poor examples. These ontologies include some hierarchical information, but they do not provide the logical relationships that would be needed to allow a software agent to make the appropriate inferences. We hope that our own ontology will successfully define the terms and relationships that we will need to allow effective searching and discovery of our services in the registry.

4 Deliverables

In cooperation with both our advisors at Lincoln and our advisor at WPI, we determined that there would be four "tangible" products delivered at the end of the project. These would be: the sensor description files, the sensor ontology, the registry/repository to store the information, and a client to allow for basic searching of the registry/repository.

4.1 Sensor Files

The first step in our project was to produce the set of files describing the sensors. Our advisors had done some research into, and requested we use, the Sensor Modeling Language (SensorML) to represent our services. The Sensor Modeling Language was developed by the Open Geospatial Consortium as a way to make a standard description of sensors. The hope of the OGC is that eventually all sensors will be described using SensorML allowing programmers to easily access the content of any of them. Currently the OGC has released documents describing the format for a SensorML file, OGC 07-000. In addition to this, the OGC has begun to develop a dictionary of terms that can be used to help describe the components, attributes, and services of most sensors. The OGC is currently applying for a URN namespace, OGC 06-166. It is in our best interest to try and follow this standard since we want to make sure that multiple users will be able to use our services.

The SensorML schema allows for highly-specific descriptions of sensors. A SensorML file can contain a diverse range of data, from basic information like a sensor's location to niche information like the physical location of each sensor on the tower. The OGC has included this breadth of information in order to facilitate SensorML's adoption as the standard XML

description of a sensor. Our test, however, just needed to have enough information to be able to be meaningfully stored in a registry, so we did not need to use the entire SensorML spec.

Our sponsors at Lincoln Laboratory gave us a (non-schema) set of XML files characterizing roughly 240 radar sensors. There were three types of radar given in the files, all of which were weather-oriented sensors (NEXRAD, CANRAD, and TDWR). For each radar defined, the files gave the call-sign (e.g., KABR), location (in latitude and longitude), elevation (in meters), tower height (in meters), declination (in degrees), beam width (in degrees), frequency (in megahertz), and information about the scanning pattern.

Given this data to encode, our SensorML boilerplate broke up into eight main parts: Name, Description, Keywords, Identifiers, Classification, Characteristics, and Location. Additionally, with more varied data, the capabilities category could also be used. In general, the keywords, identifiers, classification, and characteristics sections describe the various features of the sensor.

4.2 Ontology

The first step in providing this was finding out how to develop an ontology. There are many ways you can develop ontologies and many different tools that you can use. With so many options it can be a difficult decision. We decided, therefore, to find a simple starting point that would cover the basic concepts and concerns that we would have to cover in our ontology. We found such a source in a paper produced by Natalya F. Noy and Debora L. McGuinness from Stanford University[11].

In it, Noy and McGuinness identify a list of steps to create a good ontology:

1. Determine the domain and scope of the ontology. Answer some basic questions about the ontology's purpose.
   (a) What is the domain that the ontology will cover?
   (b) For what are we going to use the ontology?
   (c) For what types of questions should the information in the ontology provide answers?
   (d) Who will use and maintain the ontology?
2. Consider reusing existing ontologies.
3. Enumerate important terms in an ontology. This should be a list of terms that the ontology should be able to make statements about or explain to the user.
4. Define the classes and the class hierarchy.
5. Define the attributes (properties) of the classes.
6. Define the constraints on the attributes (type, cardinality, etc.)
7. Create several instances.

Some of the tasks have already been defined by our problem scope. Our ontology covers sensor characteristics, and we'll be using it to provide inference capabilities to our sensor database. The ontology will be used by those with a need for sensor data, and it will be maintained by an organization with an interest in keeping a database of sensors, like the FAA or Lincoln Lab.

Putting ourselves in the place of one of the theoretical users, we generated a set of questions that we thought would be potentially relevant queries. Later on in the project, we can use these questions as a testable measure of how well our ontology (and later, registry) performs its intended purpose. These seven questions cover a wide amount of the intended scope of our sensor database.

1. What are the radar stations in a 50 mile radius of Boston/Logan airport?
2. What does NEXRAD/CANRAD/TDWR detect?
3. What is the difference between NEXRAD and CANRAD?
4. Which radar stations will give me more detailed information on wind shear?

5. What kind of a sensor is NEXRAD/CANRAD/TDWR?
6. Who owns the NEXRAD/CANRAD/TDWR radar stations?
7. What are all of the stations owned by (someone)?

There is little work that has been done with regards to sensor ontologies. Some effort was made by NASA and the JPL to develop a sensor ontology as part of their SWEET work, but it was little more than a list of vocabulary, and wasn't really thorough enough to serve as a good starting point. The Marine Metadata Interoperability Project was (as of September 2007) looking at creating a sensor ontology, but the focus would be on marine sensors, not weather and surveillance sensors[9].

Based on the aforementioned sensor ontologies, discussions with supervisors, and our research, we came up with a list of terms that we decided would be relevant to the topic and our ultimate goals.

- Sensor
- Radar
- NEXRAD
- CANRAD
- TDWR
- latitude
- longitude
- altitude
- beam width
- frequency
- tower height
- declination
- observable type
- owner
- airports
- cities
- miles

- kilometers
- meters

Following an example in the Protégé OWL tutorial[8], we made our top-level classes the main categories of our concepts are Sensor, ObservableType, Location, Owner, and SensorType.

In order to evaluate the effectiveness of our ontology, we used the evaluation procedure recommended in *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*[7].

According to Gruber, a good ontology should exhibit the following traits: clarity, coherence, extendibility, minimal encoding bias, and minimal ontological commitment[1]. An ontology has *clarity* when its definitions and logical restrictions are unambiguous. It has *coherence* when all of its logical assertions are non-contradictory. An ontology is *extendible* when new definitions can be added with little or no modifications of already existing logical assertions. *Encoding bias* occurs when ontological decisions are made for reasons of notation or implementation, rather than purely knowledge-based reasons. Lastly, *ontological commitment* occurs when an ontology defines more than is necessary for the application at hand.

After determining the structure of our ontology, we needed to figure out how we were going to encode it. There are several schemas and projects that exist to encode ontologies[2], but OWL was recommended by our Lincoln sponsors, so that was what we used. In addition to being our sponsors' recommendation, OWL also had the advantage of being an XML-based format, which would be convenient given the robust XML support on the web today.

[1]There is a certain amount of tension between these concepts. A drive for clarity tends to increase ontological commitment, for example.

[2]KIF, CycL, and Rule Interchange Format to name a few.

To develop the ontology, we chose to use Protégé , an open source ontology editor developed at Stanford. It provides a built-in reasoner for consistency checking, visualization options, and other useful features. Protégé can natively operate on OWL, and implements everything supported in OWL-DL, so it was perfect for this purpose. We did, however, mostly use the most recent alpha version (4.0alpha as of this writing), so there was some instability. Another unfortunate flaw was the lack of reasoner support for some OWL-DL features. The power of an ontology is the ability for computers to make inferences from the ontology data, so lack of some features of that (cardinality reasoning, non-class properties) was detrimental.

4.3 ebXML Registry

Now that we have our SensorML files and our ontology, we need to make them available to the public. To do this we use a registry, and in our specific case an ebXML registry. It was the decision of our Lincoln Laboratory advisors to use an ebXML registry. Our advisors had done their own research into UDDI and ebXML (which are currently the best known registry systems) and decided on going with ebXML. Our advisors also told us that we should use the Omar registry implementation produced by the freebXML organization.

Beyond the preference of our Lincoln Laboratory advisors, we had other reasons for going with Omar. One is that it comes with two types of clients, a Java browser and a HTML Interface that can be accessed and used by most web browsers. This means that we do not have to try to write our own client to test the registry. The web browser client allows anyone to access the registry without needing to install any software on his or her own systems. Both clients allow users to sign up to become registered. This helps maintain

registry integrity, since the maintainers of the registry can control access. Once a person has become a registered user they can make changes and add new content to the registry.

Another big reason for using the Omar registry is that one of the original authors of the code, Farrukh Najmi, is doing consulting work for Lincoln Laboratory. So by going with the Omar registry we have a source of information on how the registry was created and what its features are. The registry also was designed so that it could be used with multiple databases like Derby, HSQLDB, MySQL, and Oracle Database. Along with these multiple database formats, the Omar registry can be run off of multiple server implementations. However, Tomcat does appear to be the preferred server.

The registry is our access to the world. For our data to be usable by many people we have to make sure that others can understand and use our registry. The creators of Omar took this into account by designing it around the Registry Information Model produced by the OASIS group.

OASIS stands for the Organization for the Advancement of Structured Information Standards. They are a non-profit organization that aids in the development, convergence, and adoption of open standards. OASIS produces standards for many areas such as "security, Web services, conformance, business transactions, supply chain, public sector, and interoperability within and between marketplaces"[13].

The OASIS ebXML Registry Information Model Version 3.0.1 defines the most up-to date standards that all ebXML registries should follow to be consistent. To support the Registry Information Model eighteen classification schemes were developed. These classification schemes provide extensible enumeration types to the registry. A registry could include more classifications schemes to fit their particular purpose, however these are the eighteen re-

quired canonical classification schemes that all ebXML registries must have. For our project we will need more ClassificationSchemes then just the eighteen listed here. We will also be adding the SensorType and ObservableType ClassificationSchemes discussed later in this section.

The Registry Information Model has both a hierarchical and partive relationship structure. Figure 1 only represents a subset of the metadata classes defined by the model and their partive ("has-a") relationships. Figure 2 shows the inheritance relationships ("is-a") of the information model.

Using this model any content such as documents, music, videos, or our SensorML files is considered a RepositoryItem. These items are then stored in a content repository so they can be accessed and distributed as needed. Along with our SensorML files we also have to store some metadata that we can use to describe our RepositoryItems. This data would be some sub-object of RegistryObjects or a RegistryObject itself. The RegistryObjects get stored in the registry of our ebXML implementation. The OASIS standard summarizes this concept as "ebXML Registry stores any type of content as RepositoryItems in a repository and stores standardized metadata describing the content as RegistryObjects in a registry"[5].

Figure 3 shows the library metaphor of how RegistryObjects and RepositoryItems work. On the first line you can see that we have a RepositoryItem being represented as a book. If we want to add this to our registry then we first make a RegistryObject to represent the RepositoryItem. Similarly in a library you would make a card catalog entry for the book before you put it on the shelf. The second line shows that information from inside the RepositoryItem is used to configure the RegistryObject, just like information from a book would be used to make its card in the card catalog. Information is also added to the

Table 1: The eighteen canonical classification schemes in ebRIM[5].

AssociationType	Defines the types of associations between RegistryObjects.
ContentManagementService	Defines the types of content management services.
DataType	Defines the data types for attributes in classes defined in[5].
DeletionScopeType	Defines the values of the deletionScope attribute in RemoveObjectRequest protocol message.
EmailType	Defines the types of email addresses.
ErrorHandlingModel	Defines the types of error handling models for content management services.
ErrorSeverityType	Defines the different error severity types encountered by registry during processing of protocol messages.
EventType	Defines the types of events that can occur in a registry.
InvocationModel	Defines the different ways that a content management service may be invoked by the registry.
NodeType	Defines the different ways in which a ClassificationScheme may assign the value of the code attribute for its ClassificationNodes.
NotificationOptionType	Defines the different ways in which a client may wish to be notified by the registry of an event within a Subscription.
ObjectType	Defines the different types of RegistryObjects a registry may support.
PhoneType	Defines the types of telephone numbers.
QueryLanguage	Defines the query languages supported by a registry.
ResponseStatusType	Defines the different types of status for a RegistryResponse.
StatusType	Defines the different types of status for a RegistryObject.
SubjectGroup	Defines the groups that a User may belong to for access control purposes.
SubjectRole	Defines the roles that may be assigned to a User for access control purposes.

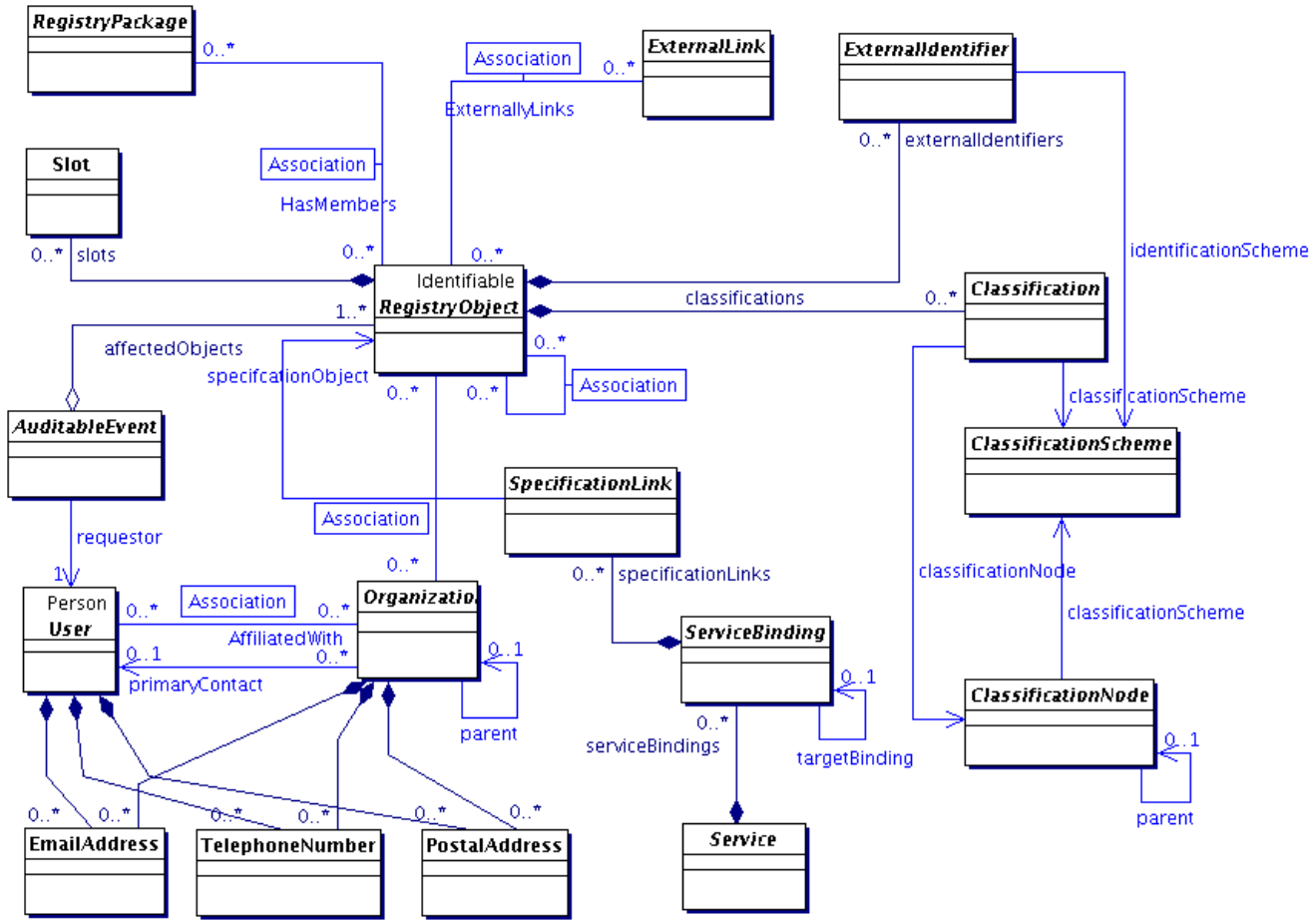

Figure 1: Information Model Relationships View[5]

RegistryObject so that the RepositoryItem it represents can be found. This is the same as defining where a book belongs by using the Dewey Decimal system. On the last line you can see that the RegistryObject is put into the registry, like a card is placed into the card catalog. The RepositoryItem is placed into the repository, like the book would be placed on the book shelf. When someone wants to find a RepositoryItem in the registry they will search through the RegistryObjects, just like a person would check the card catalog of a

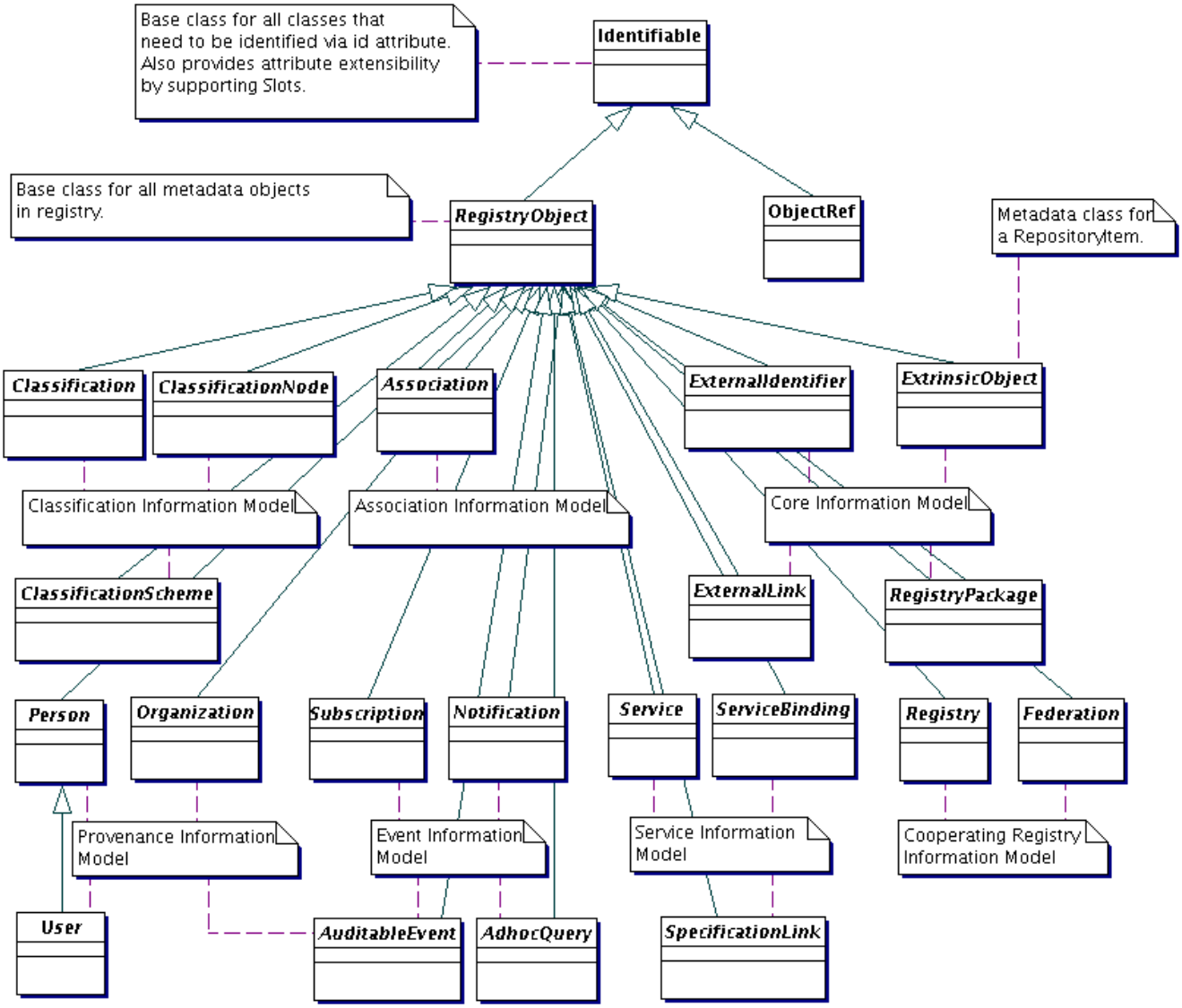

Figure 2: Information Model Inheritance View[5]

library to find a particular book.

The ebXML registry has three different kinds of users (though one user could act in multiple roles): the administrator who sets up the registry, the content publisher who adds

their information, and the user who searches to discover content. When beginning our work on the registry we became administrators to it. We had to determine what classification schemes and capabilities needed to be implemented.

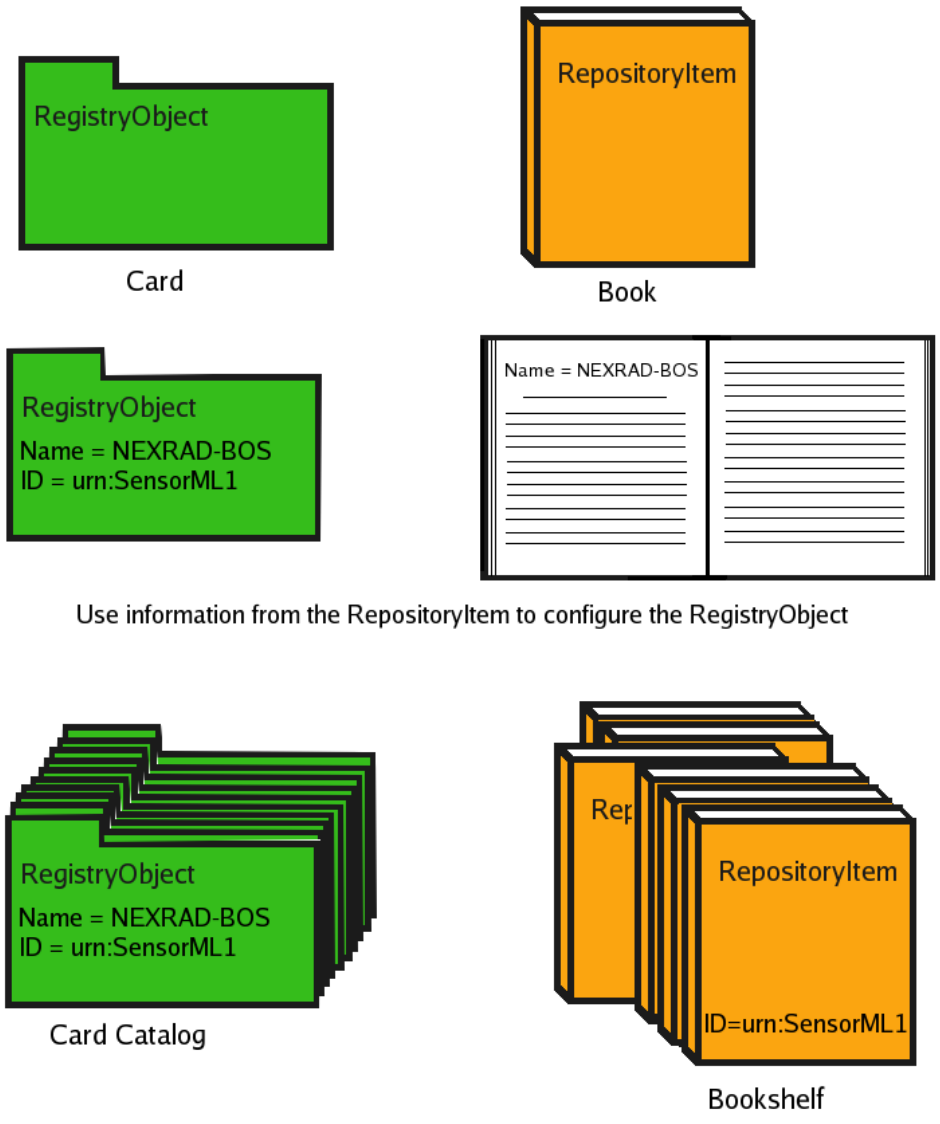

Figure 3: The registry seen as a library

We needed to define what ClassificationSchemes are needed to help represent our data. For the purposes of this project we have two different schemes that our Lincoln Laboratory advisors and we believed were necessary. These are SensorType and ObservableType. SensorType indicates if this sensor is meant for detecting weather phenomena or if it is for tracking moving aircraft. ObservableType describes what particular weather phenomena a weather-detecting sensor is meant to find, this includes phenomena like rain, wind, etc. Any SensorML file should include information on the classification of that particular sensor.

We also had to determine the objectType to use for our SensorML files. ObjectType is an attribute listed in the Registry Information Model. According to the RIM all instances of a RegistryObject must have an objectType attribute. Furthermore the value of this objectType, "must be a reference to a ClassificationNode in the canonical ObjectType ClassificationScheme"[5]. A ClassificationNode according to the Registry Information Model is an instance "used to define tree structures where each node in the tree is a ClassificationNode"[5]. This means that any of the subclasses of a ClassificationScheme in our registry is made up of ClassificationNodes. An administrator can add their own ClassificationNodes to the registry just like they can add their own ClassificationSchemes. The only difference is that the ClassificationScheme is the root of the classification tree while all descendent types are ClassificationNodes. In most instances the objectType of an object is just the name that identifies their class. The RIM lists the example that "objectType for a Classification is a reference to the ClassificationNode with code 'Classification' in the canonical ObjectType ClassificationScheme"[5].

Since our objectType must be a ClassificationNode we need to either use a pre-existing ClassificationNode or make our own. Since the default Registry Information Model does not

include any specific ClassificationNodes to represent sensors we had to make our own. We had to make this ClassificationNode a subclass of the ExtrinsicObject class. We did this because the ExtrinsicObject class is the "primary metadata class for a RepositoryItem"[5]. This means that the ExtrinsicObject class allows us to upload data into the registry. The ExtrinsicObject class already has several child types. One of these is XML. Since our SensorML files are an extension to the XML language we made our new ClassificationNode a subclass of this object. We named this new ClassificationNode SensorML so that it could be easily identified as the objectType needed to represent a SensorML object.

One of the advantages to using the XML ClassificationNode (and any of its children) is that it already supports some of the associations needed for a function of Omar that we would like to use called cataloging. In the Omar registry cataloging is the automated process that uses an extension to the XML language called XSLT. XSLT - or XSL Transformation - is a language that defines how to translate one XML document into another or to a file of a different type. In our case we want to use an XSLT file to pull some of the data out of our SensorML files and use it to create the RegistryObject.

Having our registry automatically generate this information is far better for the user than entering it in manually. First it makes the addition of SensorML files to the registry fast, simple, and consistent. The user does not have to worry about how their data should fit into the registry. The user only has to know that a SensorML file is stored as the SensorML ClassificationNode.

Our first step in establishing cataloging was the development of an XSLT file to make the transformation from our SensorML files to the Registry Information Model format. By using two example XML files, provided by our Lincoln Laboratory advisors and Farrukh

Najmi that represented a SensorML file and its RegistryObject, we developed an example of the XSLT file. This transformation took elements from the original SensorML, including the values from tags that represent keywords, identification information, description information, classification information, and information on websites that contain other information relating to the sensors and created a RegistryObject to represent our SensorML file. After developing the XSLT file we proceeded to configure the registry so it would run automatically when a SensorML file was saved under the SensorML ClassificationNode. Farrukh helped us in this area by both showing us what configuration files had to be changed and by showing us examples of what modifications produced what results. The addition of cataloging information to the registry requires that the XSLT file be uploaded to the registry as an ExtrinsicObject so it can be accessed as needed. Along with uploading the XSLT an Association (one of the ClassificationSchemes that are required by the Registry Information Model) must be made between the XSLT file and a ClassificationNode.

We associate our XSLT file to the SensorML ClassificationNode that we created using an Association called CatalogingControlFileFor. With this setup all XML files that are uploaded to the SensorML ClassificationNode and then saved are transformed into a RegistryObject. This XSLT file is stored in the registry and will be on a CD of all of our work that we turn in to our Lincoln Laboratory advisers.

It is possible to make all of these configurations to the registry through one of the client browsers. However, since configuration falls under the role of the administrator, it makes more sense to load our configurations externally. To add our XSLT file, the new SensorML ClassificationNode, Association and our new ClassificationSchemes from outside the running registry we use a build configuration defined in the file user-build.XML of the Omar code.

This target uploads the XSLT file and a configuration file, in XML, that defines our new ClassificationNode, ClassificationSchemes and Association. By using an ant target to configure the Omar registry we can modify any Omar registry (Omar version 3.1) by just changing a few files. It was Farrukh who first made us aware of the ant targets and configuration files available in Omar to get all of this data into the registry. Without this knowledge we would have added in all of this data through the client so as not to risk modifying the code of the registry.

When we first tried to use the cataloging feature of Omar we had problems. Everything had been configured but the transformation was not happening correctly. We would save a file but none of the information from the RepositoryItem would be added to the RegistryObject. It turned out that our first draft of the XSL file was not formatted correctly.

In Omar the XSLT files used in cataloging must read from two input sources. One of them is the original RegistryObject that the user creates to submit their data. The second input is the RepositoryItem the user uploads. Once we discovered this, we had to modify our XSLT file to work in this manner. A Java class in the registry called CanonicalXMLCatalogingService.Java takes the new RegistryObject and uses this as the main input for the XSLT. This class also creates a parameter whose value allows access to the RepositoryItem. Usage of this parameter allows for the reading of data from the RepositoryItem by the XSLT file. Cataloging is run using the Xalan XSLT processor. Xalan is a Java based XSLT processor that implements the Java.XML.transform interface. Once completed our cataloging feature allowed the registry to automatically generate the RegistryObjects needed to represent our SensorML data. Figure 4 shows the relationship between the SensorML ClassificationNode, its parent XML, and our XSLT cataloging file.

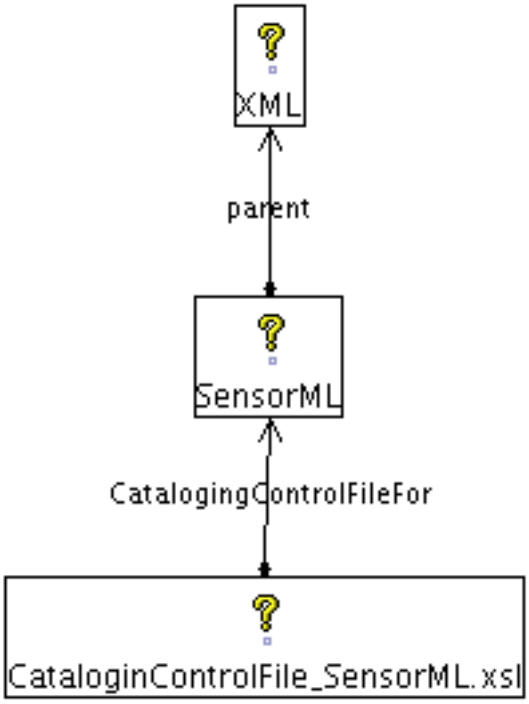

Figure 4: A diagram that can be displayed by the Java browser

Next we have two figures. Figure 5 shows the input to the XSLT file, which is a RepositoryItem of the XSLT ClassificationNode.

Figure 6 shows the result of the transformation. Note the original SensorML RegistryObject has been written over. Also the new RegistryObject still has the same link to the original RepositoryItem.

With cataloging implemented, publishing sensor data is simple. When someone wants to publish their data they first create a SensorML file to represent it. They then either access the registry through their Web browser or through the Omar Java browser. Once registered as a user of the registry one can create a new object of the SensorML ClassificationNode

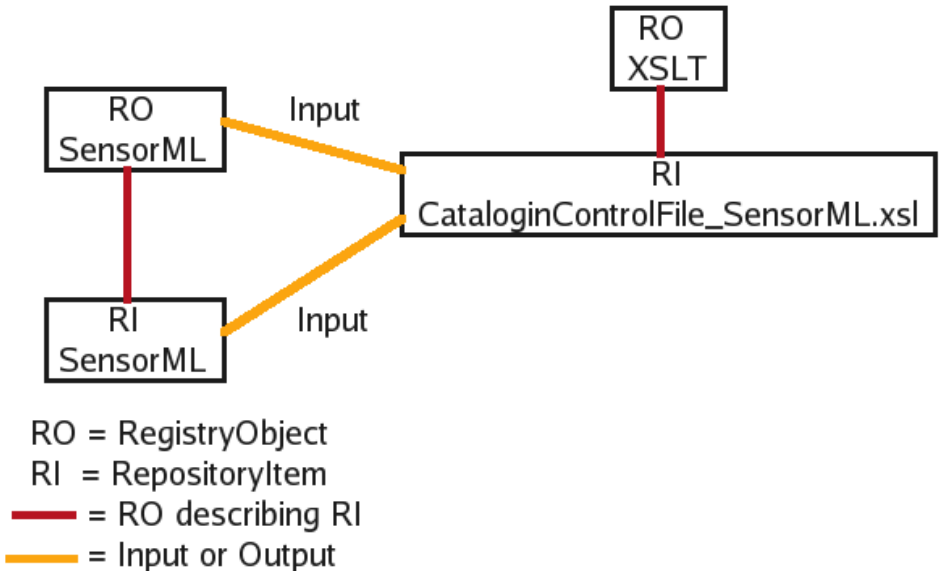

Figure 5: Input to the XSLT

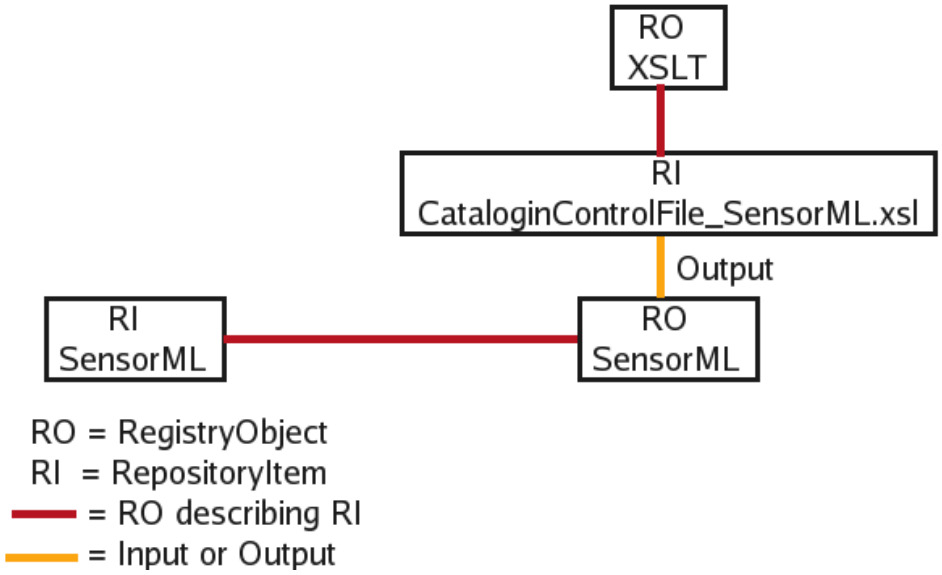

Figure 6: Output from the XSLT

type. The user just has to select their SensorML file and then save their new SensorML object. Omar will check for any associations with the SensorML ClassificationNode. Since we have a CatalogingControlFileFor association, Omar will access the listed XSLT file and run the Xalan XSLT processor. The output of this procedure is the new RegistryObject to represent the RepositoryItem. The new RegistryObject can now be discovered using the search queries made available by Omar.

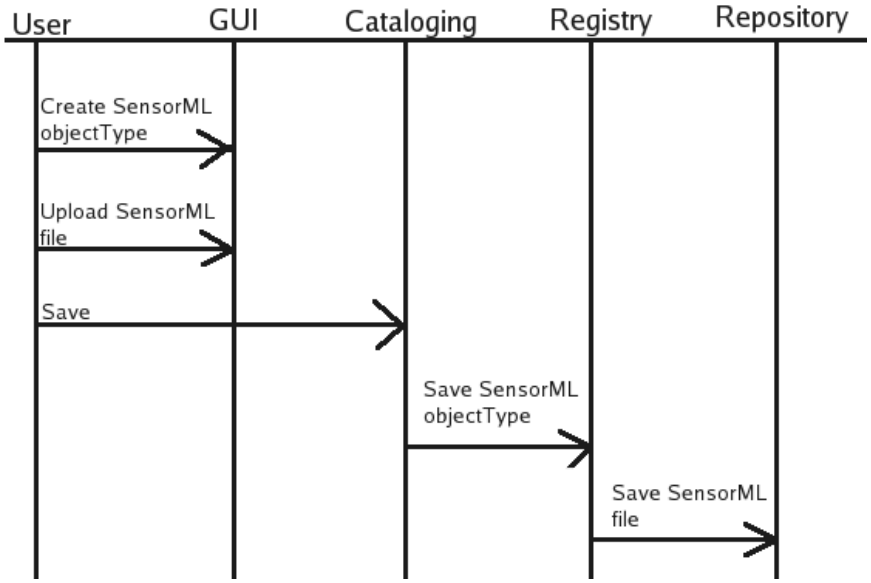

Figure 7: Submission to the Registry

Once cataloging is run, the original SensorML RegistryObject has been changed to indicate other important information such as name, description, important values, and links to other objects created in the registry to help represent this SensorML object. Take the example of a SensorML file named NEXRAD-BOS. In the NEXRAD-BOS SensorML file it defines that there is a Web site where more information can be found on the sensor. This gets represented as an ExternalLink. The SensorML file also defines that the SensorType of

NEXRAD-BOS is WeatherRadar and that it has two ObservableTypes of Rain and Wind. The NEXRAD-BOS object and the other objects that help to represent it are depicted in figure 8.

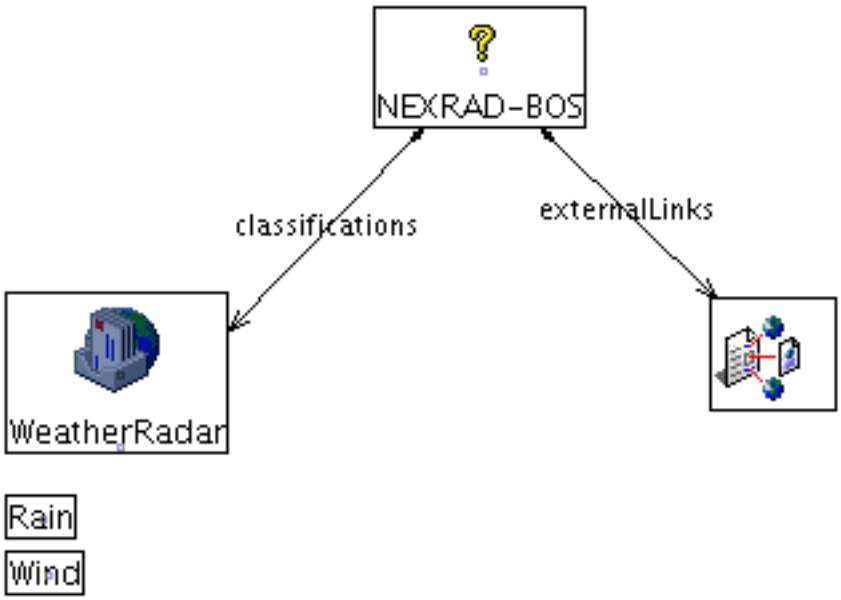

Figure 8: Object Browser in action

The Java browser and the Web page interface come with some pre-made SQL statements for searching. The default method of searching the registry is through the use of a Business Query. This query searches the Name, Description, External Links, External Identifiers, and the Classifications of the objects in the registry. For our purposes we want to make it easy for a user to find just our SensorML objects. To do this we can use a feature of the Omar registry called ad-hoc query. An administrator implementing the Omar registry can define their own SQL statements along with the parameters needed by the SQL statements. When

a client accesses an Omar registry with defined ad-hoc queries it will display the newly made query types along with fields to enter any needed parameters.

To design our own ad-hoc queries we just had to edit two files in the Omar distribution. One of them was an XML file that defined the SQL queries that would be used to search the database (SubmitObjectRequest_ AdhocQuery.XML). The other was an XML file that defined the parameters the user could enter into the client gui's to run the searches (config.XML).

We developed two of our own queries. The first of these is the query Find All Sensors. Find All Sensors is a query that will search and return all of the SensorML objects in the database using the following SQL command (please note that this query is formatted so that it can be stored in an XML file that is then loaded into the registry. The statement is then parsed and all of the syntax used by XML to interpret the command are replaced with the actual SQL syntax):

```
SELECT DISTINCT ExtrinsicObject.* FROM ExtrinsicObject WHERE
ExtrinsicObject.ObjectType=
''urn:freebXML:registry:sample:profile:sensorml:objectType:SensorML''
```

In this query ExtrinsicObject is the table that is searched in the database. To find all of the SensorML objects we select to return all of the ExtrinsicObjects where the objectType is equal to the identifier for the SensorML ClassificationNode. Since the registry can hold multiple types of data, we felt it was important to give the user the ability to get all of the SensorML objects in the registry, without having to look through all of the items stored in the registry.

The other SQL query we developed is called Find Sensors. This query just takes one parameter, called Keyword, and searches the Slot attribute of the SensorML ClassificationNode using the following SQL command (please note that this query is formatted so that it can be stored in an XML file that is then loaded into the registry. The statement is then parsed and all of the syntax used by XML to interpret the command are replaced with the actual SQL syntax):

```
SELECT DISTINCT ExtrinsicObject.* FROM ExtrinsicObject, Slot WHERE
ExtrinsicObject.ObjectType
=''urn:freebXML:registry:sample:profile:sensorml:objectType:SensorML''
and UPPER(Slot.Value) = UPPER(''\dollarsign Keyword'') and
ExtrinsicObject.Id = Slot.Parent
```

In this query, ExtrinsicObject and Slot are tables in the registries database that are searched. This search will find any ExtrinsicObject that has an objectType equal to the identifier for the SensorML ClassificationNode and whose Value field in the Slot table matches the parameter Keyword and whose Parent field of the Slot value is the Id field for the ExtrinsicObject. The types of values stored in the Slot attribute of the SensorML ClassificationNode includes any data from the SensorML file that was defined using the following tags; sml:keyword, sml:identifier, gml:name, and gml:description. This will often include short names for the sensor, long names, manufacturer names, and keywords that the creator of the SensorML file thought would be useful. One may also notice that this second query uses the function UPPER when comparing the Keyword parameter to the value field of the Slot table. This function just converts all of the letters to upper case. This way our search is not case sensitive.

Once the user has run a search using one of the available query methods the browser will display all of the matching objects from the registry (Note that usage of the Business Query searches the entire registry and is not limited to just the SensorML ClassificationNode). The user can then browse the RegistryObjects that represent the data, other objects that are associated with the RegistryObjects, and the RepositoryItems themselves.

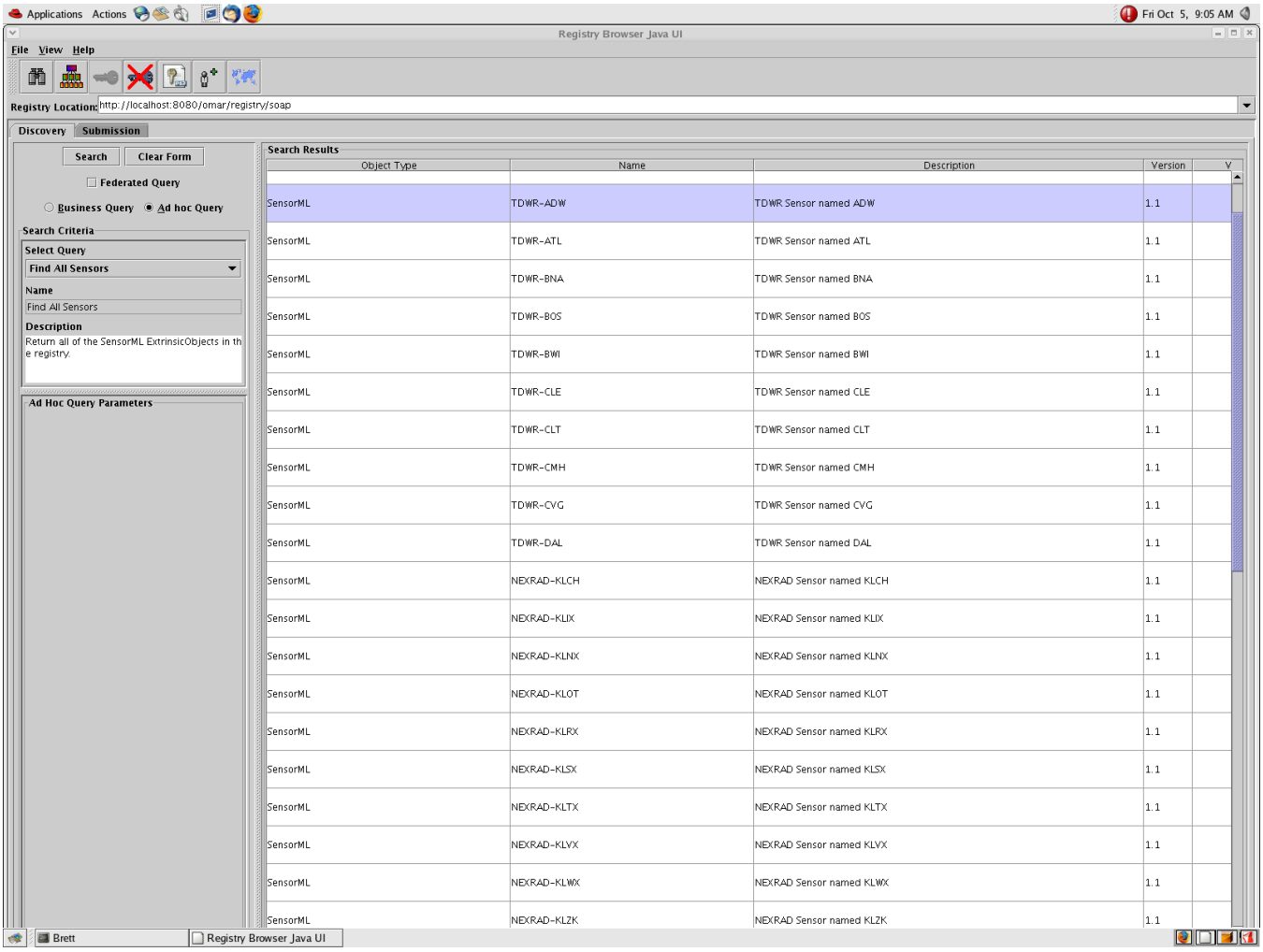

Figure 9: The Java Browser

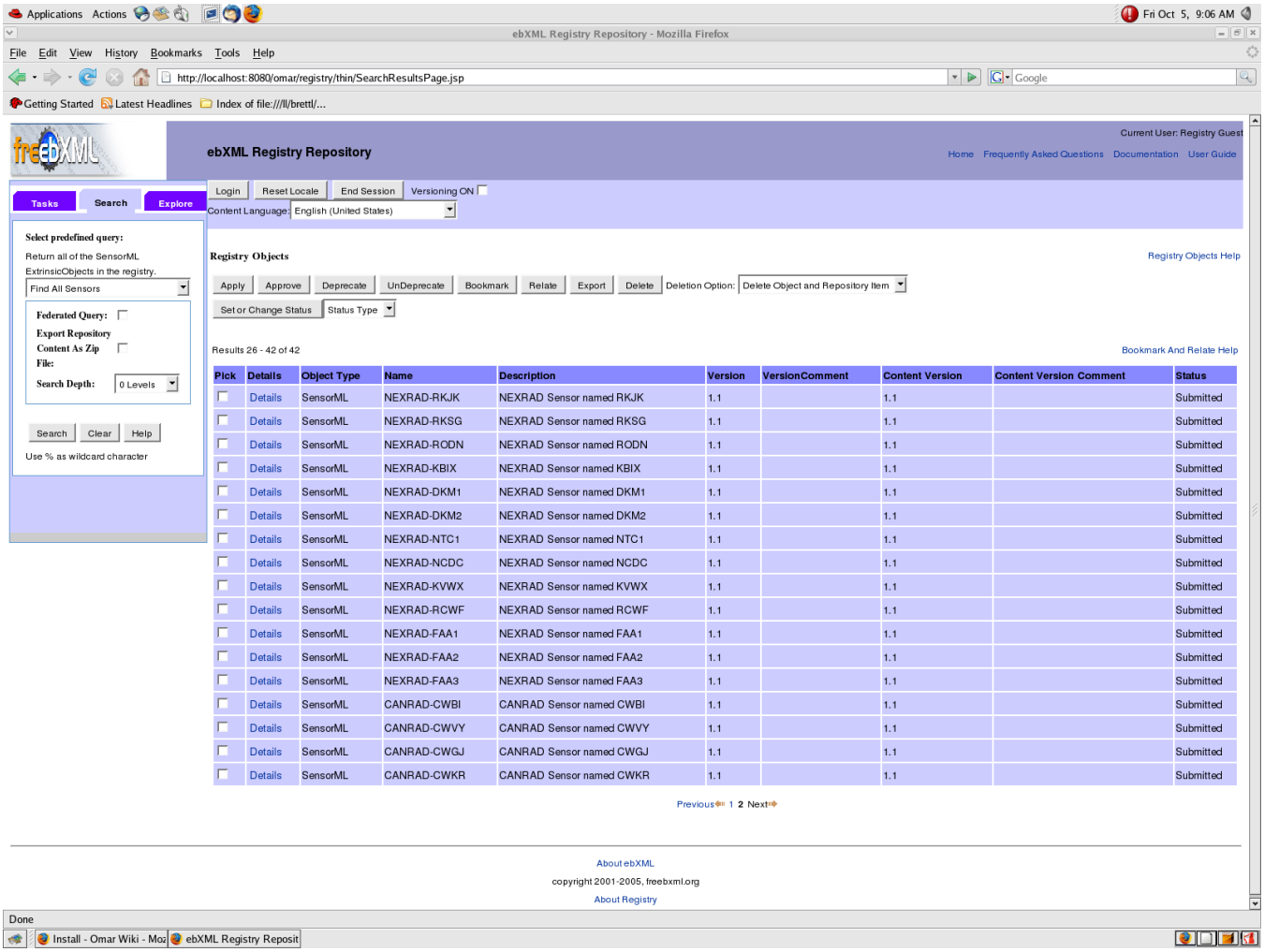

Figure 10: The Web Browser in Firefox

While making our SQL queries we ran into a problem. The problem came up when we tried to design our query to find a sensor based on location. We wanted to have the ability to enter in a location and a range around it to find the sensors. To do this requires comparison between number values for latitude and longitude. We had originally wanted this range to be some unit of measure like kilometers or miles. However we did not want to restrict the user

to one unit of measurement. Also if we tried to make a parameter to define a unit then we would need multiple conditional statements and mathematical equations for conversion right in our SQL statement. We then thought of having our range be just in degrees of latitude and longitude to make a box around any particular area. Unfortunately data entered in to the ExtrinsicObject is stored as a string. Normally we would just cast this value to a decimal and then do our comparison. However it turns out that there is a known bug in the current release of the Omar registry that prevents this.

Omar uses a parser to break up SQL statements. The purpose of the parser was to remove certain predicates of the SQL statement when its parameter was not given. The idea is that if a search query has many predicates and the user only enters the parameters for a few of them the parser can remove the unneeded predicates to improve efficiency. Unfortunately this parser does not support all of the SQL language. All of the casting techniques we tried were unrecognizable and the query would never be sent to the database.

We consulted Farrukh on this problem and he informed us that there was a workaround in the current development version of the registry. Unfortunately, the Lincoln Laboratory firewall blocked our access to this data. While it still would have been possible to get a copy of this version of Omar and get it running, time was short. Instead, we decided to go with a different solution for finding sensors based on location. We made a Java program that would take a SensorML file, find the latitude and longitude coordinates in it, and place them into a KML file, which is an extension to the XML language that represents three dimensional geospatial data. We can then upload these KML files into the program Google Earth and actually see where the sensors are. While this is not as desirable as having the registry tell you what sensors fall within a certain range, it was the best we could do with the time we

had left.

Another problem that we ran into with ad-hoc queries were how the client handled parameters. Apparently when a query uses no parameters, like our Find All Sensors query, it is possible for the Java Browser client to fail to run the query. We seem to experience this problem when we first try to use a query with a parameter and then try to use one without them. This is most likely due to how we configured our queries though as opposed to a bug in the code. While this error occurred in the Java Browser it did not appear at all in the HTML interface to the registry. Since it is possible to run searches using our ad-hoc queries through a Web browser without any problems we decided not to spend a great deal of time attempting to fix the problem. If we had more time we would have looked into this more, but this was just not the case.

Despite these problems, we do have the registry and queries operational. A user can access the registry and use either the built in queries or our ad-hoc queries to run searches. When you select one of the returned objects, you have the option to look at the RepositoryObject. If the object is an ExtrinsicObject or a child type of ExtrinsicObject, then you can also choose to look at the RepositoryItem. When these options are selected, the Web browser displays the XML files from the registry. From here these files can be saved to the user's system.

The data contained in the RegistryObject should be sufficient for the user to determine if a particular sensor fits their needs. However as stated earlier a user cannot run searches based on location. To solve this problem a user can download a Java program from the registry that uses XSLT to convert SensorML files into KML files. Once the user has selected the sensors that they would like to check they could run the Java program and create the KML

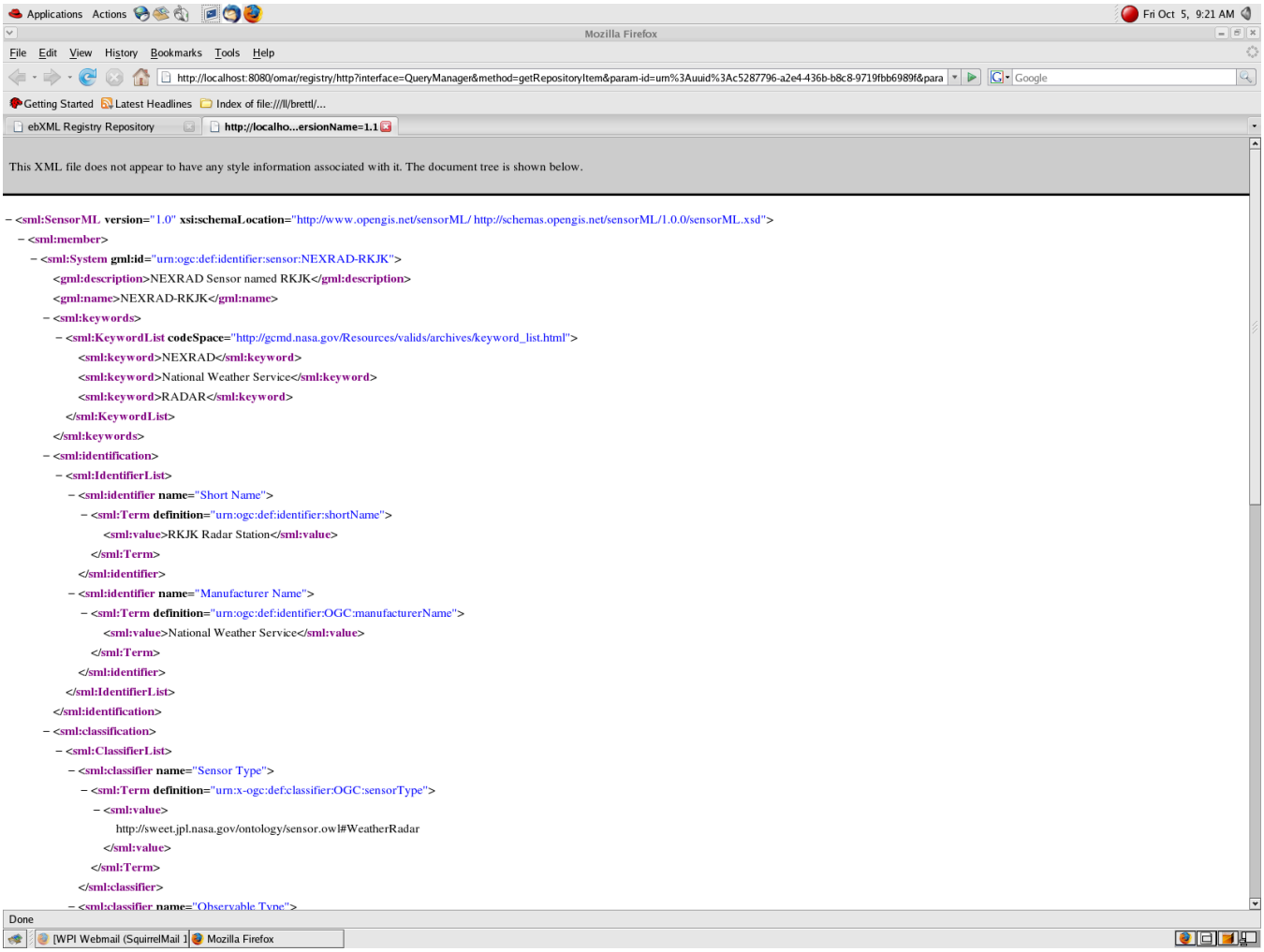

Figure 11: A SensorML file from the registry

files needed. Once completed these KML files can be loaded into Google Earth and used to display the actual location of the sensors. With this visual representation the user should be able to determine which of the displayed sensors fall within their desired range.

We tested to see if our registry could actually return the values that we wanted. We did this by first uploading 41 sensors to the registry. Forty of these were our own SensorML

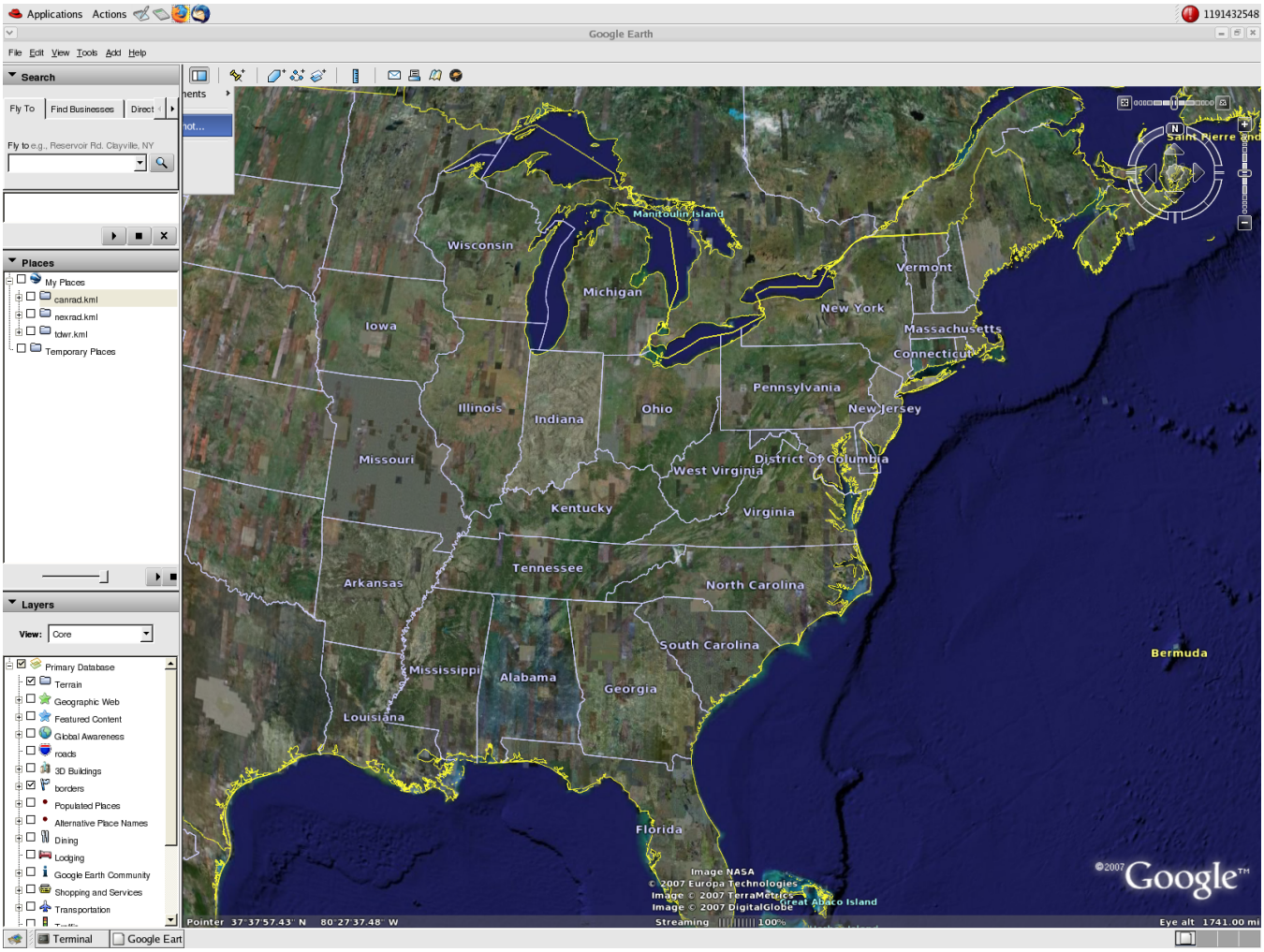

Figure 12: Google Earth without sensor overlay

files while another one was a test file given to us by our Lincoln Laboratory advisors. The breakdown of these sensors was 10 TDWR, 27 NEXRAD, and 4 CANRAD. For our first test we ran the Find All Sensors query. This did indeed return all 41 of our SensorML objects.

For our next tests we looked up many of the different values contained in the Slot attributes for our SensorML RegistryObjects. We then ran searches using these strings and

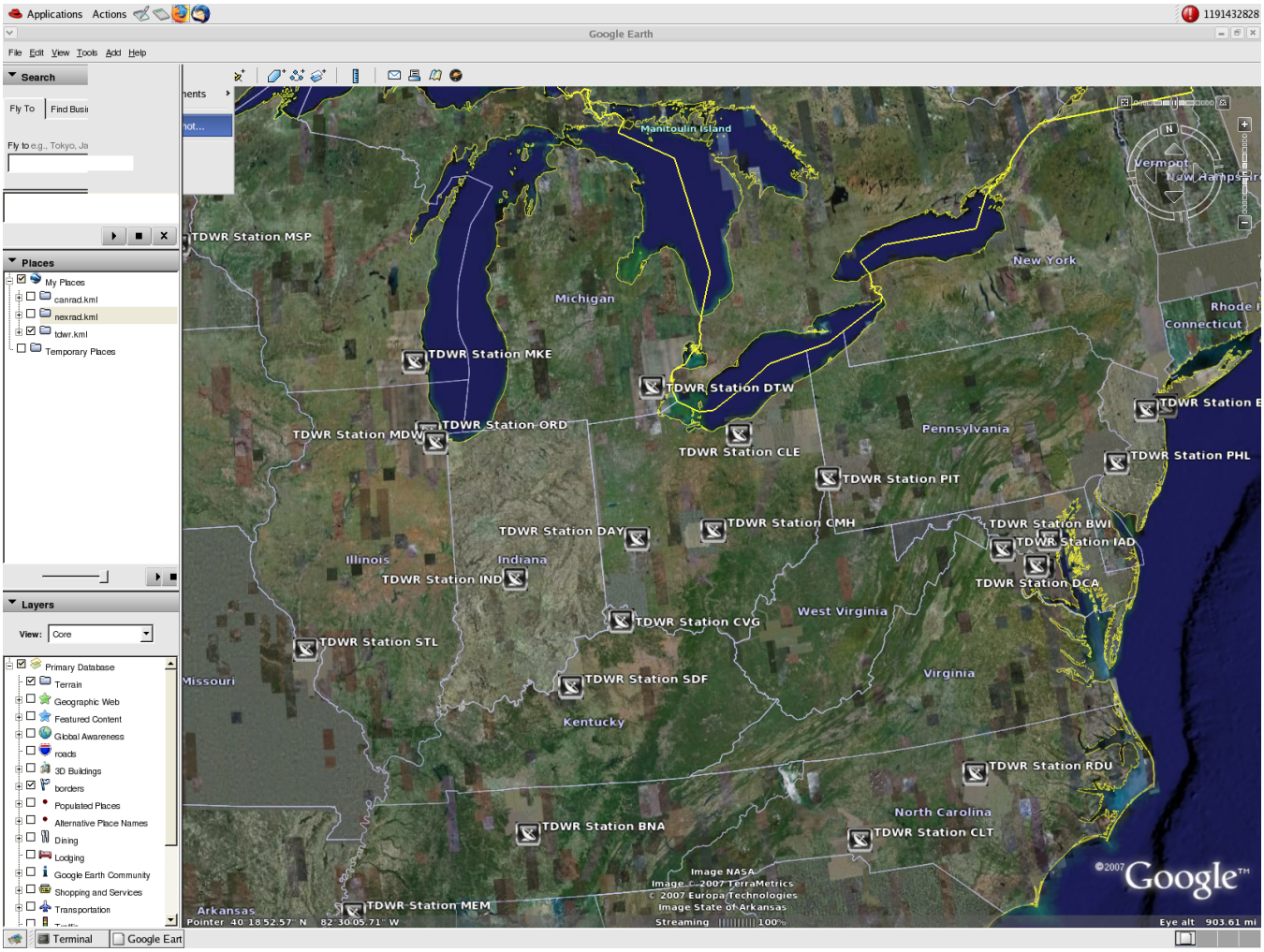

Figure 13: Google Earth without TDWR sensors overlaid

checked to see if the returns contained all the correct sensors. The next tables show these test runs. The first table checks for keywords that should return multiple values. The second table uses keywords that should only return one particular sensor.

There are a few things in the registry we would change and add on to if we had the time. We were told by Farrukh that it is possible to map an ontology written in OWL Light onto

Table 2: Test results for multi-sensor queries

	NEXRAD Sensors		TDWR Sensors		CANRAD Sensors	
Keyword	Found	Actual	Found	Actual	Found	Actual
NEXRAD	27	27	0	0	0	0
TDWR	0	0	10	10	0	0
CANRAD	0	0	0	0	4	4
Radar	27	27	10	10	4	4
National Weather Service	26	26	0	0	0	0
Canada	0	0	0	0	4	4

Table 3: Test results for single-sensor queries

Keyword	Found	Should be
FAA	NEXRAD-BOS	NEXRAD-BOS
S-band	NEXRAD-BOS	NEXRAD-BOS
NWS	NEXRAD-BOS	NEXRAD-BOS
WSR-88D	NEXRAD-BOS	NEXRAD-BOS
Weather	NEXRAD-BOS	NEXRAD-BOS
Nexrad Weather Radar	NEXRAD-BOS	NEXRAD-BOS
KLZK Radar Station	NEXRAD-KLZK	NEXRAD-KLZK
RKJK Radar Station	NEXRAD-RKJK	NEXRAD-RKJK
ADW Radar Station	TDWR-ADW	TDWR-ADW
CVG Radar Station	TDWR-CVG	TDWR-CVG
DAL Radar Station	TDWR-DAL	TDWR-DAL
CWVY Radar Station	CANRAD-CWVY	CANRAD-CWVY
CWBI Radar Station	CANRAD-CWBI	CANRAD-CWBI

the registry. So we would likely have tried to make a version of the ontology in OWL Light so we could map it directly into the classification setup in the ebXML registry. This way even though we could not use our ontology in the registry directly, we could still use its concepts and ideas.

We also would have liked to take more time to try to correct our query problems. Since we know that a workaround does exist in the current development version of Omar we would have obtained it and tried to develop our ad-hoc queries for checking latitude and longitude values.

5 Conclusions and Future Work

The three deliverables for this MQP were completed. We developed a set of SensorML files to describe our sensors. Along with this we also made a utility that can take the XML files that Lincoln Laboratory currently uses to store sensor information and automatically generate our SensorML files. An example ontology that defines some of the basic concepts for our sensors was completed and produces no errors or logical inconsistencies. The Omar registry was successfully configured to support our SensorML information, including cataloging, ad-hoc queries, ClassificationSchemes and ClassificationNodes. A Java program was also developed to take our SensorML files and produce a KML file that can be loaded into Google Earth to show where the sensor is. All of the work that we have generated (including programs, SensorML files, the ontology, and Omar configuration files) are on a CD that we have given to our Lincoln Laboratory advisors. This CD also includes help files that give instructions on how to use these programs and how to configure the Omar registry.

We had originally wanted to use our competency questions for our ontology to measure the accuracy of the responses from our registry. We were unable to get a direct correlation between the wording of these questions and the search capabilities of the registry. However, the answer to these questions can still be examined.

1. What are the radar stations in a 50 mile radius of Boston/Logan airport?

   To find a radar station within a certain area of a location a user can take the SensorML files from the registry and then use our Java program to convert them into KML files. Once in this format the user can run Google Earth and see if the radar stations are in the range the user wants.

2. What does NEXRAD/CANRAD/TDWR detect?

   To answer this question the user can look at the RegistryObject that describes a NEXRAD, CANRAD, or TDWR sensor and look at the classification field. The SensorType classification will tell the user if the sensor is for detecting weather or aircraft. Also the ObservableType classification will tell the user what specific weather phenomena the sensor detects.

3. What is the difference between NEXRAD and CANRAD?

   The user can look at the RegistryObjects for a NEXRAD and a CANRAD sensor and observe the differences.

4. Which radar stations will give me more detailed information on wind shear?

   This question can be answered by examining the ObservableType classification of any SensorML object.

5. What kind of a sensor is NEXRAD/CANRAD/TDWR?

   This question can be answered by examining the classification values of the RegistryObject that represents these sensors.

6. Who owns the NEXRAD/CANRAD/TDWR radar stations?

   To answer this question the user can examine the Slot attribute of the RegistryObjects and look for information on the manufacturer.

7. What are all of the stations owned by (someone)?

   Unfortunately this question cannot be answered by our registry. We had originally wanted to include other objects in our registry that would have represented a person or organization that could own the sensors. We would then of had an Association between that owner and all of the sensors that they owned. Then by searching an

owner you could discover all of the sensors that they own. We ran out of time in our project before we were able to look into this area.

There is further work that can be added to this project. Our ontology could be expanded upon. So far the ontology we produced is an example that just defines some of the terms and logic for the sensors that we used. The ontology could be expanded include other types of sensors. This would include other types of weather radar, tracking radar, sonar sensors, thermal sensors, and other types of sensor technology.

While the Omar registry is not designed to use an ontology directly, it is possible to map ontological concepts onto the registry. This would require configuring the ClassificationSchemes, ClassificationNodes, Associations, and other aspects of the Registry Information Model to reflect the same concepts and ideas as defined by our ontology. This way a user would not have to look at our ontology separately, they could just access the registry and see how all of the data fits together.

To determine a sensor's location, a user can convert the SensorML files into KML and display them in Google Earth. It is worth looking into enhancing the registry to allow searches based on location. This way the user can get this information right from the registry and would not have to use another program like Google Earth. We had originally wanted to be able to define a location whose latitude and longitude coordinates was known to the registry and specify the range around that spot to find sensors. Unfortunately due to the problems discussed when attempting to make an ad-hoc query to search by location we could not get this done for our project. However, this would be a perfect addition for others to enhance our work.

For ourselves, this MQP has given us valuable experience in different fields. Service

Oriented Architecture, Semantic Web, and ontologies are all fields that are becoming big in computer science. Our research showed us that many people are looking to these technologies for better solutions and new possibilities. The work we accomplished on this project has given us introductory experience to all of these areas and could prove to be very helpful to us in our future careers.

The members of Group 43 at Lincoln Laboratory will benefit from this project in many ways. At present few of the groups at Lincoln Laboratory have done research in these areas. Group 43 knows that for future projects they will want to use the technologies that we have researched for this project. Our MQP has provided them with the bibliographical references to documents we read and has produced functioning samples for them. The problems that we ran up against are now known to our advisers, and will cause less confusion in the future. From here Group 43 can have future MQP groups expand upon our work, or could expand upon the work themselves. Whatever path the members of Group 43 choose they will have this project to help guide them in their future work.

A Glossary

Aviation Routine Weather Report (METAR) Hourly weather data containing observations on temperature, dew point, wind, precipitation, cloud cover, visibility, and barometric pressure. METAR comes from the French, "message d'observation météorologique régulière pour l'aviation."

Canadian Radar (CANRAD) A network of radar devices that is roughly the Canadian equivalent of NEXRAD.

Derby An open source database project developed by Apache. The Derby database is implemented completely in Java.

Dictionary A listing of the concepts in a domain along with their meanings.

Domain The conceptual space that an ontology covers.

Electronic Business using Extensible Markup Language (ebXML) "A modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet."[12]

Extensible Markup Language (XML) A markup language that allows users to define their own tags and schema.

Extensible Stylesheet Language Transformation (XSLT) An XML stylesheet that defines how to reformat one XML document into another.

FreebXML A free and open source implementation of the ebXML standard intended to facilitate the adoption of ebXML.

HSQL Database (HSQLDB) An open source relational database engine written in Java.

Institute of Electrical and Electronic Engineers (IEEE) A non-profit organization that seeks to enhance technology by creating standards and providing resources to the professional community.

Keyhole Markup Language (KML) An XML schema developed by Google to create overlays on their Google Earth program.

MySQL An open source relational database management system that uses SQL for querying and management.

Next Generation Radar (NEXRAD) A network of roughly 150 hi-res doppler radars used to detect precipitation and wind.

Ontology A data model that represents concepts and their relationships within a domain.

Open Geospatial Consortium (OGC) An organization dedicated to the creation of standards and specifications that facilitate the access and presentation of geographical or other location-based data.

Oracle Database A relational database management system produced by Oracle.

Organization for the Advancement of Structured Information Standards (OASIS) A non-profit consortium that aids in the development, convergence, and adoption of open standards.

Protégé A free, open-source ontology editor developed by Stanford Medical Informatics at the Stanford University School of Medicine.

Resource Description Language (RDF) An XML schema that allows the user to make statements about information by using subject-predicate-object expressions.

Semantic Web The web as a collection of data presented in both traditional form and in a form that outlines conceptual relationships in a computer-readable way.

Sensor Model Language (SensorML) An XML schema developed by the OGC to create a standard for defining sensors.

Structured Query Language (SQL) A language for requesting information from a database.

Surface Synoptic Observations (SYNOP) Coded weather forecasts sent out every six hours from manned and automated weather stations.

Taxonomy A set of hierarchical classes dividing the concepts of a domain into groups.

Terminal Doppler Weather Radar (TDWR) A radar sensor installed at many major airports to detect local windshear and precipitation.

Tomcat A Java-based servlet developed by Apache.

Universal Modeling Language A graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

Web Ontological Language (OWL) An XML schema developed to express ontologies.

Xalan An XSLT processor for transforming XML documents in to HTML, text, or other XML document types.

B SensorML

The information contained in the SensorML files is based on consultations with our advisors regarding what data they would like to see in these files.

```
<?XML version="1.0" encoding="UTF-8"?>
<sml:SensorML XMLns:sml="http://www.opengis.net/sensorML/1.0"
              XMLns:swe="http://www.opengis.net/swe/1.0"
              XMLns:gml="http://www.opengis.net/gml"
              version="1.0"
              XMLns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.opengis.net/sensorML/
                 http://schemas.opengis.net/sensorML/1.0.0/sensorML.xsd">
  <sml:member>
    <sml:System gml:id="urn:ogc:def:identifier:sensor:NEXRAD-KBOX">
```

The introductory section of the SensorML files is basically set in stone. These top-level tags contain little information unique to the sensor, with the exception of the OGC identifier term.

```
      <gml:description>NEXRAD Sensor named KBOX</gml:description>
      <gml:name>NEXRAD-KBOX</gml:name>
      <sml:keywords>
        <sml:KeywordList
          codeSpace="http://gcmd.nasa.gov/Resources/valids/archives/keyword_list.html">
          <sml:keyword>NEXRAD</sml:keyword>
          <sml:keyword>National Weather Service</sml:keyword>
          <sml:keyword>RADAR</sml:keyword>
        </sml:KeywordList>
      </sml:keywords>
```

The keyword section consists simply of a list of terms related to the sensor. These terms

need not be thematically related, but are merely words which, if searched for, would want to bring this sensor as a result.

```
<sml:identification>
  <sml:IdentifierList>
    <sml:identifier name="Short Name">
      <sml:Term definition="urn:ogc:def:identifier:shortName">
        <sml:value>
          KBOX Radar Station
        </sml:value>
      </sml:Term>
    </sml:identifier>
    <sml:identifier name="Manufacturer Name">
      <sml:Term definition="urn:ogc:def:identifier:OGC:manufacturerName">
        <sml:value>
          National Weather Service
        </sml:value>
      </sml:Term>
    </sml:identifier>
  </sml:IdentifierList>
</sml:identification>
```

The identification section contains names that can be used to identify the sensor, along with the names of related organizations.

```
<sml:classification>
  <sml:ClassifierList>
    <sml:classifier name="SensorType">
      <sml:Term definition="urn:x-ogc:def:classifier:OGC:sensorType">
        <sml:value>
          http://sweet.jpl.nasa.gov/ontology/sensor.owl#WeatherRadar
        </sml:value>
      </sml:Term>
    </sml:classifier>
```

```
    <sml:classifier name="ObservableType">
      <sml:Term definition="urn:x-ogc:def:classifier:OGC:observableType">
        <sml:value>
          http://www.link.to.our.ontology/sensor.owl#Weather
        </sml:value>
      </sml:Term>
    </sml:classifier>
  </sml:ClassifierList>
</sml:classification>
```

The classification section is used for sensor aspects - that is, the categories in to which the sensor would fall. For example, what kind of sensor is it? What does it watch?

```
<sml:characteristics>
  <swe:DataRecord definition="urn:ogc:def:property:physicalProperties">
    <swe:field name="towerHeight">
      <swe:Quantity definition="urn:ogc:def:property:height">
        <swe:uom code="m" />
        <swe:value>30.0</swe:value>
      </swe:Quantity>
    </swe:field>
    <swe:field name="beamWidth">
      <swe:Quantity definition="urn:ogc:def:property:width">
        <swe:uom code="deg" />
        <swe:value>0.95</swe:value>
      </swe:Quantity>
    </swe:field>
    <swe:field name="frequency">
      <swe:Quantity definition="urn:ogc:def:property:frequency">
        <swe:uom code="mhz" />
        <swe:value>2700.0</swe:value>
      </swe:Quantity>
    </swe:field>
    <swe:field name="declination">
      <swe:Quantity definition="urn:ogc:def:property:declination">
        <swe:uom code="deg" />
```

```
            <swe:value>0.0</swe:value>
          </swe:Quantity>
        </swe:field>
      </swe:DataRecord>
    </sml:characteristics>
```

The characteristics section lists those properties of the sensor which are fixed (range-less). Additionally, there may be a capabilities section following which lists sensor properties which have ranges (survivable temperature, range of view).

```
    <sml:location>
      <gml:Point gml:id="SYSTEM_LOCATION"
                 srsName="urn:ogc:def:crs:EPSG:6.1:4329">
        <gml:coordinates>-71.138 41.956 70.4088</gml:coordinates>
      </gml:Point>
    </sml:location>
  </sml:System>
</sml:member>
</sml:SensorML>
```

Finally, the location section is merely the GML-based coordinate of the sensor.

C SensorML to ebRIM XSL Transformation

```
<?XML version="1.0"?>
<!--================================================================
Date:   10/4/2007

About:  This XSLT file allows the transformation from a Sensor Modeling
    Language format of XML document to an XML document that fits
    into the Registry Information Model used by ebXML registries.

================================================================-->


<xsl:stylesheet version="1.0"
        XMLns:xsl="http://www.w3.org/1999/XSL/Transform"
        XMLns:sml="http://www.opengis.net/sensorML/1.0"
        sml:schemaLocation="http://www.opengis.net/sensorML/1.0.0/sensorML.xsd"
        XMLns:gml="http://www.opengis.net/gml"
        gml:schemaLocation="http://www.opengis.net/gml/3.1.1/base/gml.xsd"
        XMLns:rim="urn:oasis:names:tc:ebXML-regrep:xsd:rim:3.0"
        rim:schemaLocation="http://docs.oasis-open.org/regrep/v3.0/schema/rim.xsd"
        XMLns:xlink="http://www.w3.org/1999/xlink">

  <xsl:output method = "XML" indent = "yes"/>

  <xsl:strip-space elements = "*"/>
  <xsl:param name="repositoryItem"></xsl:param>

  <xsl:template match="/">
    <xsl:variable name = "RI" select = "document($repositoryItem, .)"/>

    <xsl:variable name="ID">
      <xsl:call-template name="substring-after-last">
        <xsl:with-param name="input">
         <xsl:value-of select="//rim:ExtrinsicObject/@id"/>
        </xsl:with-param>
        <xsl:with-param name="substr">:</xsl:with-param>
```

```
      </xsl:call-template>
    </xsl:variable>



<!--================================================================
Get the gml ID information.
=================================================================-->

    <xsl:param name="ExtrinsicObjectID"
               select="normalize-space($RI//sml:System/@gml:id)"/>


    <rim:RegistryObjectList XMLns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="urn:oasis:names:tc:ebXML-regrep:xsd:rim:3.0
                                    file:///ll/brettl/Desktop/rim.xsd">

      <rim:ExtrinsicObject
        id="{//rim:ExtrinsicObject/@id}"
        objectType = "{//rim:ExtrinsicObject/@objectType}">
        <xsl:if test="//rim:ExtrinsicObject/@mimeType">
          <xsl:attribute name="mimeType">
            <xsl:value-of select="//rim:ExtrinsicObject/@mimeType"/>
          </xsl:attribute>
        </xsl:if>



<!-- ================================================================
     Retrieve all of the keywords from the KeywordList
     node in the source file.
=================================================================-->
        <rim:Slot name="urn:x-ogc:def:identifier:slotName:keywords">
          <rim:ValueList>
            <xsl:for-each select="$RI//sml:keyword">
              <rim:Value><xsl:value-of select="."/></rim:Value>
            </xsl:for-each>
          </rim:ValueList>
```

```
        </rim:Slot>


<!--================================================================
Get the identifiers from the
Identification section of the source file
=================================================================-->


        <xsl:for-each select="$RI//sml:identifier">
          <xsl:param name="IdDefinition0"
                     select="normalize-space(.//sml:Term/@definition)"/>

          <rim:Slot name="{$IdDefinition0}">
            <rim:ValueList>
              <rim:Value><xsl:value-of select="./sml:Term/sml:value"/></rim:Value>
            </rim:ValueList>
          </rim:Slot>

        </xsl:for-each>



<!--================================================================
Get the gml:name from our source file.
=================================================================-->
        <xsl:param name="Name" select="normalize-space($RI//gml:name)"/>
        <rim:Name>
          <rim:LocalizedString charset="UTF-8" value='{$Name}'/>
        </rim:Name>


<!--================================================================
Get the gml:description from our source file.
=================================================================-->
        <xsl:param name="Description" select="normalize-space($RI//gml:description)"/>

        <rim:Description>
```

```
          <rim:LocalizedString charset="UTF-8" value='{$Description}'/>
        </rim:Description>



<!--================================================================
    Get the classification information from our source file.
=================================================================-->
        <xsl:if test="$RI//sml:classifier">
          <xsl:for-each select="$RI//sml:classifier">
            <xsl:param name="Value"
                       select="normalize-space(./sml:Term/sml:value)"/>
            <xsl:param name="ClassificationID"
                       select="normalize-space(./sml:Term/@definition)"/>


            <!-- Get the portion of the ClassificationID string
                           that comes after the last ':' character -->
            <xsl:variable name="Type">
              <xsl:call-template name="substring-after-last">
                <xsl:with-param name="input">
                  <xsl:value-of select="$ClassificationID"/>
                </xsl:with-param>
                <xsl:with-param name="substr">:</xsl:with-param>
              </xsl:call-template>
            </xsl:variable>

            <!-- Get the portion of the Value string
                            that comes after the last '#' character -->
            <xsl:variable name="Type2">
              <xsl:call-template name="substring-after-last">
                <xsl:with-param name="input">
                  <xsl:value-of select="$Value"/>
                </xsl:with-param>
                <xsl:with-param name="substr">#</xsl:with-param>
              </xsl:call-template>
            </xsl:variable>
```

```
            <!-- Write our informtion to the source file -->

            <rim:Classification id="urn:SensorML:classification:{$ID}:{$Type}:{$Type2}"
                      classificationScheme="{$ClassificationID}"
                      nodeRepresentation="{$Type2}"
                      classifiedObject="{//rim:ExtrinsicObject/@id}">
              <rim:Name>
                <rim:LocalizedString charset="UTF-8" value='{$Type2}'/>
              </rim:Name>
            </rim:Classification>

          </xsl:for-each>
        </xsl:if>
      </rim:ExtrinsicObject>



<!--================================================================
Get the external link information from our source file.
=================================================================-->

      <xsl:if test="$RI//sml:onlineResource">
        <xsl:param name="ExternalURI"
             select="normalize-space($RI//sml:onlineResource/@xlink:href)"/>
        <xsl:param name="Description1"
   select="normalize-space($RI//sml:documentation/sml:Document/gml:description)"/>
        <rim:ExternalLink id="urn:SensorML:ExternalLink:{$ID}"
                          externalURI="{$ExternalURI}">
          <rim:Slot name="urn:x-ogc:def:sml:date">
            <rim:ValueList>
              <rim:Value><xsl:value-of select="$RI//sml:date"/></rim:Value>
            </rim:ValueList>
          </rim:Slot>

          <rim:Slot name="urn:x-ogc:def:sml:format">
            <rim:ValueList>
              <rim:Value><xsl:value-of select="$RI//sml:format"/></rim:Value>
```

```
            </rim:ValueList>
          </rim:Slot>

          <rim:Description>
            <rim:LocalizedString value="{$Description1}"/>
          </rim:Description>

        </rim:ExternalLink>
        <rim:Association id = "urn:SensorML:onlineInformation:{$ID}"
   associationType = "urn:oasis:names:tc:ebXML-regrep:AssociationType:ExternallyLinks"
            sourceObject = "urn:SensorML:ExternalLink:{$ID}"
            targetObject = "{//rim:ExtrinsicObject/@id}"/>
      </xsl:if>


    </rim:RegistryObjectList>
  </xsl:template>




<!--====================================================================
    substring-after-last
    This template finds the last portion of a string that appears
    after the substr parameter. This code was taken from the website
    http://proquest.safaribooksonline.com/0596009747/XSLTckbk2-CHP-2-SECT-4
=====================================================================-->
  <xsl:template name="substring-after-last">
    <xsl:param name="input"/>
    <xsl:param name="substr"/>

    <!-- Extract the string which comes after the first occurrence -->
    <xsl:variable name="temp" select="substring-after($input,$substr)"/>

    <xsl:choose>
      <!-- If it still contains the search string the recursively process -->
      <xsl:when test="$substr and contains($temp,$substr)">
        <xsl:call-template name="substring-after-last">
```

```
          <xsl:with-param name="input" select="$temp"/>
          <xsl:with-param name="substr" select="$substr"/>
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$temp"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

D SensorML to KML XSL Transformation

```
<?XML version="1.0" encoding="ISO-8859-1" ?>
<xsl:transform version="1.0"
XMLns:xsl="http://www.w3.org/1999/XSL/Transform"
XMLns:sml="http://www.opengis.net/sensorML/1.0"
XMLns:gml="http://www.opengis.net/gml">
<xsl:output method="XML" indent="yes" />
<xsl:template match="/">
  <KML XMLns="http://earth.google.com/KML/2.2">
    <Folder>
      <Style id="sensor">
        <IconStyle>
          <Icon>
            <href>sensor.png</href>
          </Icon>
        </IconStyle>
      </Style>
      <Placemark>
        <name>
          <xsl:value-of select="sml:SensorML/sml:member/sml:System/gml:name"/>
        </name>
        <styleUrl>#sensor</styleUrl>
        <Point>
          <coordinates>
            <xsl:call-template name="replace-string">
              <xsl:with-param name="from" select="' '"></xsl:with-param>
              <xsl:with-param name="to" select="','"></xsl:with-param>
              <xsl:with-param name="text"
   select="sml:SensorML/sml:member/sml:System/sml:location/gml:Point/gml:coordinates">
              </xsl:with-param>
            </xsl:call-template>
          </coordinates>
        </Point>
      </Placemark>
    </Folder>
  </KML>
```

```
</xsl:template>

  <!-- reusable replace-string function -->
  <!-- Taken from http://aspn.activestate.com/ASPN/Cookbook/XSLT/Recipe/65426 -->
  <xsl:template name="replace-string">
    <xsl:param name="text"/>
    <xsl:param name="from"/>
    <xsl:param name="to"/>

    <xsl:choose>
      <xsl:when test="contains($text, $from)">

        <xsl:variable name="before" select="substring-before($text, $from)"/>
        <xsl:variable name="after" select="substring-after($text, $from)"/>
        <xsl:variable name="prefix" select="concat($before, $to)"/>

        <xsl:value-of select="$before"/>
        <xsl:value-of select="$to"/>
        <xsl:call-template name="replace-string">
          <xsl:with-param name="text" select="$after"/>
          <xsl:with-param name="from" select="$from"/>
          <xsl:with-param name="to" select="$to"/>
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$text"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:transform>
```

References

[1] Accenture. Service oriented architecture for revenue organizations. `http://www.accenture.com/NR/rdonlyres/652DBCB1-AB49-4EF7-A01D-0BFDA5A97%310/0/SOAflyerv4.pdf`. Accessed Sept. 1, 2007.

[2] Alesso, H. P., and Smith, C. F. 2005. *Developing Semantic Web Services*. Perters, Natick, MA.

[3] Antoniou, G., and van Harmelen, F. 2004. *A Semantic Web Primer*. MIT Press, Cambridge, MA.

[4] Bieberstein, N. 2007. *Service Oriented Architecture Compass: Business Value, Planning and Enterprise Roadmap*. Courier, Westford, MA.

[5] Breininger, K., Najmi, F., and Stojanovic, N., (Eds.). February 2007. *ebXML Registry Information Model*, 3.0.1. OASIS ebXML Registry Technical Committee.

[6] Chu, X., and Buyya, R. 2007. Service oriented sensor web. In *Sensor Network and Configuration: Fundementals, Standards, Platforms, and Applications*, N. P. Mahalik, ed. Springer-Verlag, Germany, 51–74.

[7] Gruber, T. R. 1993. Toward principles for the design of ontologies used for knowledge sharing. Tech. Rep. KSL 93-04, Knowledge Systems Laboratory, Stanford University.

[8] Horridge, M., Knublauch, H., Rector, A., Stevens, R., and Wroe, C. 2004. *A Practical Guide to Building OWL Ontologies Using the Protégé -OWL Plugin and CO-ODE Tools*. The University of Manchester.

[9] Marine Metadata Interoperability Project. Plan for oceanographic devices ontology. http://marinemetadata.org/examples/mmihostedwork/ontologieswork/observa%bleont/ontdevices. Accessed Sept. 11, 2007.

[10] Moinuddin, M. January 2007. An overview of service-oriented architecture in retail. Tech. rep., Microsoft Corporation.

[11] Noy, N. F., and McGuinness, B. L. Ontology development 101: A guide to creating your first ontology. http://protege.stanford.edu/publications/ontology_development/ontology1%01-noy-mcguinness.html. Accessed Sept. 9, 2007.

[12] OASIS. About ebxml. http://www.ebxml.org/geninfo.htm. Accessed Sept. 9 2007.

[13] OASIS. Who we are - faq. http://www.oasis-open.org/who/faqs.php. Accessed Oct. 1 2007.

[14] Open Geospatial Consortium. Ogc web services, phase 3. http://www.opengeospatial.org/projects/initiatives/ows-3. Accessed Sept. 4, 2007.

[15] Salam, A. F., and Stevens, J. R. 2007. *Semantic Web Technologies and E-Business*. Integrated Book Technology, Hershey, PA.

[16] Viezzer, M. December 2000. Ontology in ai. http://www.cs.bham.ac.uk/~mxv/report2/node5.html. Accessed Sept. 9, 2007.